Ethan Jones

C Compiler to WebAssembly

Computer Science Tripos – Part II

Churchill College

7th May 2021

Declaration

I, Ethan Jones of Churchill College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

I am content for my dissertation to be made available to the students and staff of the University.

Signed *Ethan Jones*

Date 7th May 2021

Acknowledgements

I would like to thank my project supervisor, Dr Timothy Jones, for his continued support while developing the project and for proofreading this dissertation and giving feedback.

Additionally, I would like to thank Alex Vanlint for his feedback on this dissertation.

Proforma

Candidate Number:	**2326C**
Project Title:	**C Compiler to WebAssembly**
Examination:	**Computer Science Tripos – Part II, July 2021**
Word Count:	**11927** *
Line Count:	**13902** **
Project Originator:	Dr Timothy Jones
Project Supervisor:	Dr Timothy Jones

Original Aims of the Project

The original aim of this project was to create an optimising compiler from a subset of C targeting WebAssembly. The compiler was to be built in TypeScript, allowing its use on the web. Existing libraries would only be used to help with lexing and parsing, and the rest of the compiler, including code generation and producing WebAssembly, was to be written. Extensions included supporting compiling and linking multiple files, adding support for additional language features and adding more complex optimisations.

Work Completed

I have implemented an optimising C to WebAssembly compiler in TypeScript, including a preprocessor, lexing and parsing (using existing libraries), transformation into an intermediate representation, linking, code generation and optimisations. I have met the project's success criterion and implemented the extensions, including multiple-file support, additional C features and complex optimisations, including partial-redundancy elimination. The performance of my compiler is evaluated against Emscripten, an existing C to WebAssembly compiler, and native programs, using a custom testing framework.

Special Difficulties

None.

* This word count was computed by `detex introduction.tex preparation.tex implementation.tex evaluation.tex conclusion.tex | tr -cd [:space:][:alnum:] | wc -w`

** See Appendix A

Contents

1 Introduction

The goal of my project was to create an optimising compiler from a subset of C targeting WebAssembly. This involved implementing each of the typical compiler stages – lexing and parsing, transforming into an intermediate representation, code generation and optimisation. I successfully met my success criterion and implemented my planned extensions, allowing compiling and linking multiple files and adding more advanced optimisations.

1.1 Motivation

The primary motivation behind this project was to further my understanding of compilers beyond the Tripos, particularly the Part IB Compiler Construction course and the Part II Optimising Compilers course.

However, WebAssembly is also an especially exciting compilation target, as it is a relatively new platform with unusual design choices. It is unlike both lower-level native architectures, such as x86, and the majority of existing higher-level targets, such as the JVM's bytecode.

I chose to write the compiler in TypeScript, which compiles down to JavaScript. Modern web browsers support both WebAssembly and JavaScript, creating some interesting use cases where my compiler is used to enable local compilation and execution of C code within the browser. While writing the implementation, I created several in-browser demonstrations using this functionality to show the output of different compiler stages. In addition to precompiling C programs for use on the web, it could be used to build interactive code challenges fully contained in the browser or online tools for learning C.

Finally, I chose to compile the majority of C89[1], with some C99 features, as C is commonly used for writing high-performance and low-level programs. Additionally, to ensure the compiler could run real-world programs, I didn't want to compile a more complicated language where it would only be feasible to implement a small subset of language features.

1.2 WebAssembly

WebAssembly (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine, designed as a portable compilation target[2]. It was created to enable the execution of other programming languages in web browsers without the issues of cross-compiling to JavaScript (JS). At runtime, modules are compiled down to native instructions, giving it near-native performance. The key uses cases are integrating existing code written in other languages into web applications and increasing the performance of algorithms currently implemented in JS. Additionally, WebAssembly is increasingly being used outside the browser[3].

WebAssembly has a few limitations by design to make it safer, making it interesting to target. For example, the stack only stores basic values and is stored in a separate memory space from the memory accessible by the module, preventing arbitrary access. There are also no arbitrary jumps, and instead, structured control flow is enforced. This means that branch instructions can only jump to parent structured instructions (`if`, `block` and `loop`), each with different behaviour.

1.3 Existing Work

Emscripten[4] is the most commonly used solution for compiling C or C++ into WebAssembly. It builds upon LLVM and compiles LLVM bitcode into either JavaScript or WebAssembly.

Cheerp is another open-source C/C++ compiler for the web[5], targeting JavaScript, asm.js (a subset of JS designed as a compiler target[6]) or WebAssembly, and also builds upon LLVM.

Both of these existing projects are large and designed to be run locally. By comparison, my compiler is much smaller (although also less feature-rich) and designed to run natively inside web browsers. I am not aware of any existing C to WebAssembly compilers written in JavaScript/TypeScript.

2 Preparation

This chapter describes the techniques, tools and practices used, and contains high-level descriptions of the implemented optimisations.

2.1 Starting Point

Before this project, I had used C in some small projects, learnt TypeScript for an internship and had no experience with WebAssembly. While researching the project and planning the proposal, I read some of the WebAssembly specifications[7] and looked into the JavaScript WebAssembly Interface[8]. I also started looking into available lexers and parsers that suited my project.

My project is a standalone compiler, only depending on two libraries for lexing and parsing, as these tasks were not the focus of the project. When implementing partial support for the C standard library, I took and modified many standard-library functions from existing open-source implementations (see Appendix B for details and licences). Finally, some of the benchmarks used to evaluate my project are existing workloads (see Appendix G).

2.2 Requirements Analysis

My success criterion set out in my project proposal was:

"The project will be a success if it compiles a single-file program written in a common subset of the C language into WebAssembly and executes it correctly."

This criterion set a relatively simple requirement of creating a working compiler that can execute a subset of C. To achieve this, implementing the typical compiler stages was needed, as outlined in my introduction. To evaluate my project, I set out comparing the performance against other systems, so creating an automated benchmarking system was also useful.

My first extension was implementing support for compiling and linking multiple files, and this added requirements of an additional (limited) preprocessor stage and a linking stage. It also required implementing or finding implementations of various standard-library functions, as these are widely linked against.

My proposal also set out various C features that I planned to exclude, due to their substantial added complexity. Adding these was listed as an extension, but as the project progressed, it proved challenging to find compatible programs, so I had to modify my requirements and the target subset of C. Some of these features are now fully supported, and others have limited support, as initially envisaged.

The final requirements were from my extension of implementing more advanced optimisations. Here, I had to find suitable optimisations to perform and algorithms implementing these optimisations.

2.3 Software Engineering Techniques

2.3.1 Development Model

I chose to use the waterfall model as I was creating a new codebase from scratch and knew the requirements for each module at the start of the project. I implemented the primary stages of the compiler in order, except creating helper functions for WebAssembly output. This was performed early on in the project, as planned in my project proposal, as it required carefully reading the WebAssembly specification, giving me a better idea of the end goal. (These helpers were only used by various unit tests until the code-generation stage was implemented later.) Once the main stages were complete, I integrated the additional stages required for the extensions.

These stages could be written sequentially as each had a clearly defined output. However, when changing the requirements to support additional C features, minor modifications were necessary to the earlier stages and were implemented following an iterative model.

2.3.2 Libraries

Jison-gho

The C standard includes a summary of the grammar, and there is an online Yacc grammar version[9]. Therefore, when choosing a parser library, the key feature was compatibility with Yacc-like grammar files. The generated parser being small was also desirable to keep the compiler's size small, increasing usability from the browser. Additionally, it would keep the parser reasonably understandable if any parser bugs did arise.

I initially chose Jison, which calls itself "Bison in JavaScript"[10]. GNU Bison is compatible with Yacc[11], and Jison generated a working parser on the first attempt. Furthermore, the produced parser was relatively small and well documented. I later switched to use Jison-gho, a fork of Jison[12], which added useful features for actions in the grammar and produced a more optimised and thoroughly documented parser.

I also considered using ANTLR v4[13] (which can generate JavaScript) and antlr4ts[14] (generates TypeScript). However, both produced much larger parsers that additionally required large runtime libraries. ANTLR v4 was also hard to use from TypeScript due to being a JavaScript project without separate type definitions, and antlr4ts produced a parser incompatible with the latest TypeScript available at the time.

Moo

Jison-gho can also generate a lexer, but having the lexer in a separate TypeScript file instead of the grammar file enforces abstraction between the parser and the lexer. I knew this abstraction would be useful for later implementing `typedef` support, as C is not strictly context-free and lexical feedback is required to differentiate between identifiers and custom types[15].

I chose Moo as it is small (4KB bundled), fast, uses standard regular expressions for defining tokens and tracks line numbers[16]. It works by transforming the regular expressions into a single regular expression, allowing heavy optimisation by the JS runtime.

WebAssembly

Instead of using an existing library, I developed my own minimal set of functions to create WebAssembly modules to keep the compiler small. This was also a useful process for understanding the WebAssembly specification.

2.3.3 JS Runtime

Node.js was used to develop the project and is used to run the automated tests and build the demos, which were tested in Google Chrome. Both of these utilise V8 as their JavaScript and WebAssembly engine[17], meaning that feature support and performance characteristics are almost identical. Furthermore, Chrome is the most popular web browser[18], and Node.js offers npm, a packager manager, making it easy to manage the required tool and library dependencies.

When running WebAssembly in V8, a two-stage pipeline is used[19]. First, the module is compiled with Liftoff, a simple one-pass compiler that emits native code for each WebAssembly instruction. Since it compiles the entire module to native code in a single pass, it has very high throughput but can only perform limited optimisations. Once this step is done, the module starts executing, enabling large WebAssembly modules to be loaded and used quickly. The second step is to use TurboFan to recompile every function and then swap in the optimised versions. TurboFan is a multi-pass compiler that performs many advanced optimisations like strength reduction and function inlining.

However, runtime flags can be used to ensure that only Liftoff (`--liftoff --no-wasm-tier-up`) or Turbofan (`--no-liftoff --no-wasm-tier-up`) is used. When benchmarking performance in my evaluation, I tested with only Liftoff or Turbofan enabled to avoid the run-to-run variance in the TurboFan compilation time.

Some of the build and testing scripts utilise features specific to Node.js but, once built, the compiler uses standard ECMAScript 2020, so it should be compatible with other runtimes. I have performed limited testing in Firefox, and the demos work as expected.

2.3.4 Tools

Git and GitHub

My project's source is stored in a git repository, and each commit has a meaningful message. A public GitHub repo was used as an off-site backup, and GitHub Actions were used to run the tests and build and deploy the demos to GitHub Pages automatically after each commit.

AVA and nyc

Automated testing was used to ensure my project worked as expected and that new changes didn't break old code. I chose AVA as the test runner as it expects tests to be written using modern JavaScript, includes TypeScript definitions out of the box and enforces writing atomic tests, which it runs concurrently[20].

Additionally, I used nyc to measure test code coverage[21]. It is simple to use, only requiring the test command to be prefixed with `nyc`, and is recommended by AVA.

ESLint and TypeScript ESLint

ESLint was used to ensure a consistent style in the source code and to perform basic static analysis[22]. TypeScript ESLint is a plugin to ESLint, enabling TypeScript support and adding type-aware checks[23]. These were chosen as they easily integrate with my JS IDE of choice, WebStorm, and the main alternative is TSLint, which has been deprecated by the authors in favour of TypeScript ESLint[23].

Webpack

Webpack is a JavaScript module bundler[24] and was used to create JS bundles and simple HTML files for each of the demo webpages. It was also used to bundle the project's source code into a single JS file for use as a library.

Backup Plan

Both my project's source and the dissertation's LaTeX source are backed up in GitHub repos. Weekly backups were also made to an external drive, and my desktop performs hourly off-site backups to the cloud. In case of hardware failure, I planned to use my laptop or the MCS machines, both of which had the required software installed.

2.4 Optimisations

The optimisations that I have implemented are all mentioned in the Part II Optimising Compilers course, with the course also featuring algorithms for performing some of the optimisations. However, unlike that course, my implementations run directly on the stack-based WebAssembly code instead of on a three-address code intermediate representation (IR). I chose to do this as I added the optimisations after I had already completed the code-generation stage for my success criterion. Adding in an extra IR before the code-generation stage would have required a substantial reworking of the existing IR and the compiler backend.

Peephole Optimisations

Peephole optimisation is the process of scanning over the generated code and checking for specific patterns of instructions that can be replaced. Various peephole optimisations are implemented, mostly designed to clean up certain aspects of the generated WebAssembly where it is easier to do it as a second pass later. For example, unused WebAssembly `block` instructions are removed and replaced with their contents.

Dead-code Elimination

Dead-code elimination is the removal of code that only produces results that aren't used and which doesn't have any side effects. To implement this for stack-based code, you trace which instructions produce and consume each stack value and recursively check the stack values used by instructions with side effects. Unneeded instructions are then either removed or replaced with `drop` instructions (which remove an item from the stack) to keep the stack consistent.

2.4.1 Data-flow Optimisations

Data-flow analyses are carried out over a control-flow graph (a graph containing all possible paths of control flow through a function). Each analysis relies on a set of data-flow equations over the individual nodes in the control-flow graph and finding the solution to these equations, which can be performed iteratively.

Copy Propagation

Reaching-definition analysis is a forwards data-flow analysis that constructs a mapping between definitions of locals within a function and when they are used. Using this, you can find usages of local variables where there is only one possible definition of the local. If a definition assigns a constant to a local, then definite usages of that definition can be replaced with the constant directly, which is called constant propagation. If a definition assigns a different local y to the local x and the same local definitions are valid at a usage, the use of x can be replaced with y, called copy propagation. This is particularly useful in cleaning up code generated by other optimisation passes and removing redundant locals.

Local Allocation

WebAssembly allows each function to have a near unlimited number of locals, but reducing the number can help optimise the code produced by simpler runtimes. This can be achieved by using live-variable analysis, a backwards data-flow analysis, to find which variables are live (storing a result that may be needed later) at each point in the function. From this, a clash graph can be constructed, which contains each local as a node and edges between any simultaneously live locals. Finding an optimal allocation is then equivalent to colouring the graph using as few colours as possible, which is NP-complete, so heuristics are commonly used.

Live-range Splitting

Live-range splitting is the process of splitting function locals that are reused but could instead be separate locals, potentially improving local allocation. This can be achieved by using a forwards data-flow analysis, where a new local is allocated for each definition, and then locals are merged each time multiple paths through the control-flow graph meet.

Partial-redundancy Elimination

Partial-redundancy elimination (PRE) combines multiple other data-flow analyses (common-subexpression elimination and loop-invariant code motion) into one and is only mentioned briefly in the Optimising Compilers course. It eliminates re-computation of expressions where only some paths through the flow graph will have already computed the value by inserting extra computations in any other branches and then removing the now-redundant calculation. This is a relatively complex optimisation and needs the results of several data-flow analyses in order to do it safely and correctly. I implemented it using an algorithm presented in an article called "*A simple algorithm for partial redundancy elimination*" by V.K. Paleri[25], which involves performing four different data-flow analyses.

2.4.2 Interprocedural Optimisations

The previous optimisations are all intraprocedural, meaning that they run on each individual function. Interprocedural optimisations instead analyse and transform the entire program.

Function Inlining

Function inlining is the process of replacing a function call with the body of the target function, eliminating the function-call overhead. It can also allow for more optimisations in the target function after inlining, for example, copy propagation of constant arguments. This is achieved by calculating the call graph (mapping function calls to their target functions) and then deciding which functions should be inlined. Short target functions are candidates due to the relatively larger function-call overhead, and functions only used once are also good candidates.

2.5 Summary

This chapter has described the background of the project and the requirements. I also set out the implementation process used, in addition to the tools and library dependencies. The next chapter describes how I used these to build my compiler.

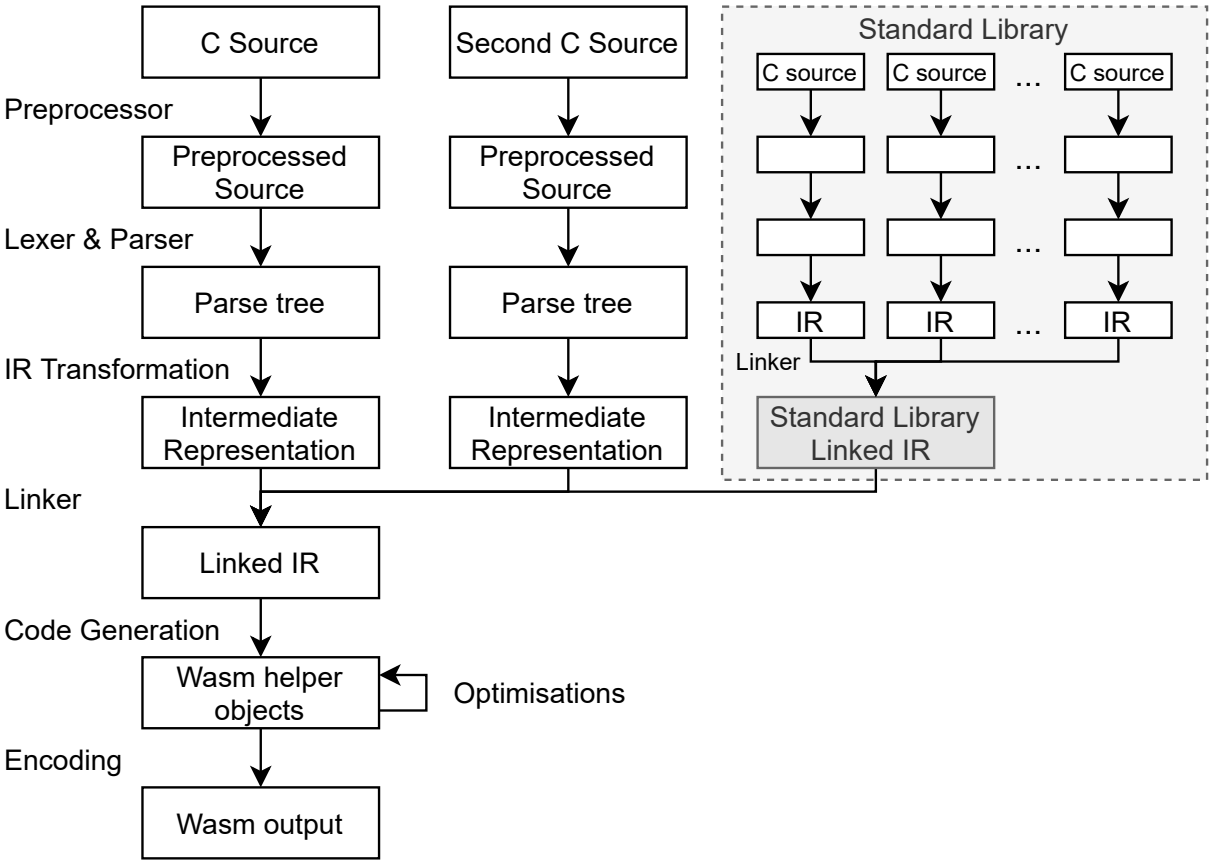
3 Implementation

This chapter presents my implementation, which I call c2wasm, a C to WebAssembly compiler written in TypeScript.

3.1 Compilation Overview

The diagram below shows an example of the compilation process involving two source files being linked against the standard library. Each source file is preprocessed, then lexed and parsed into a parse tree. Next, the parse tree is validated and transformed into the intermediate representation (IR), which is similar in structure but has added information, such as types. All the IRs and the linked standard library are then linked together. The code-generation stage then creates objects roughly corresponding to the structures (e.g. instructions, functions) in the WebAssembly output. The optimisation passes then run and modify these objects in-place. Finally, the helper objects are serialised into the actual WebAssembly module.

The linked standard library is cached at runtime and is created similarly, with each file being preprocessed, parsed and transformed into IR before being linked together.



3.2 Repository Overview

The table below gives an overview of the structure of key files and folders in the project repository. Each compiler stage has its own source folder, apart from the linker, which is implemented in a single file. Code in the `src/c_library/impl/libraries/` and `tests/benchmark/*/` folders was taken from existing projects, which are listed with their licences in Appendix B and Appendix G respectively.

`c2wasm/`	
`src/`	Source folder
`preprocessor/`	C preprocessor
`parsing/`	Parsing stage: lexer, parser setup and parse tree classes
`gen/`	Generated JS from parser generator
`c_grammar.jison`	Parser grammar
`ir/`	IR stage: classes to represent expressions, statements, etc.
`transform/`	Functions that transform the parse tree into the IR
`wasm/`	WebAssembly output helpers
`generation/`	Code-generation stage using IR and Wasm helpers
`optimisation/`	Optimisation stage
`flow/`	Flow optimisations
`interprocedural/`	Interprocedural optimisations
`c_library/`	C standard library support
`impl/`	C source and header files
`libraries/`	Modified implementations from existing libraries
`runtime/`	Runtime TypeScript enabling standard-library functions
`compile.ts`	Contains the main `compile` function
`linker.ts`	Simple linker implementation that operates on the IR
`tests/`	Automated project unit tests
`benchmark/`	Automated benchmarks
`demos/`	Demo webpages
`.eslintrc.json`	Linter configuration
`package.json`	Project dependencies and build scripts
`tsconfig.json`	TypeScript configuration
`webpack.config.js`	Webpack configurations to build demos and bundle JS

3.2.1 Building

The npm package manager is used to build and run the project. Once `npm install` has been run to install the dependencies, `npm test` will run the tests, `npm run lint` will run the linter and `npm run build` will use webpack to build the demos and bundled JS. Additionally, `npm run parser` will regenerate the parser from the grammar file. These commands and the project's dependencies are defined in the `package.json` file.

3.3 Preprocessor

The first step in the compilation pipeline is my C preprocessor, which supports the directives and macros needed to compile the test programs. This includes support for `#define` macros (with or without parameters), `#if` directives for conditional compilation and `#include` directives, which are crucial for using header files.

Parsing is achieved by recursively checking the start of each line using regular expressions and manual tracking for nesting of `#if` definitions and brackets in macro arguments. The expression bodies of `#if` directives are processed using the C lexer and parser, which are then evaluated recursively. Values are represented as JavaScript BigInts (primitive arbitrary-sized integers introduced in ECMAScript 2020[26]), as all values are meant to be integers, and the width of the integer type is meant to be the widest integer type known to the compiler.

3.4 Lexing and Parsing

The second stage is taking the preprocessed string input and converting it into a parse tree.

The token classes are defined in a JS object, where each entry represents the name of the class and either the string literal it matches (used for operators, e.g. `+=`) or a regular expression (used for identifiers and constants). C keywords (e.g. `if`, `break` and `struct`) are handled by Moo's keywords helper, which helps ensure the longest-match principle applies.

The parser grammar is defined inside a `.jison` file and was modified from an existing Yacc grammar[9] to customise the language support and add actions to the syntactic rules. Jison can automatically generate a parse tree representing each node as an array, but using this tree would be difficult as the grammar rules would be needed to understand what each array index represented. Therefore, I instead defined a TypeScript class hierarchy representing the parse tree and added corresponding actions in the grammar. This provides a layer of separation from the rules and simplifies processing the parse tree as each node is typed, and its children are accessed through meaningfully named fields. Grammar actions are also used to flatten parts of the parse tree, including the lists of specifiers and qualifiers in C declarations.

The parser generated by Jison is written in JavaScript and is wrapped in a TypeScript class, which sets up the parser and provides callbacks for calling the Moo lexer instance. It also provides a hook called by a grammar action for every `typedef` declaration, which is used to track which identifiers are being used for types, enabling basic `typedef` support.

(For more details on the chosen libraries, see preparation section 2.3.2.)

3.5 Intermediate Representation

The Intermediate Representation (IR) is structured following the C specification, with classes representing each type of declaration, statement, expression and data type. Unlike the parse tree, it is not constrained by the structure of grammar rules, allowing simplified representations of complex structures like declarations. However, the overall structure is quite similar, so parse trees are transformed by recursively analysing the parse tree and then building up the IR representation. This process also checks that the parse tree represents valid C code, including enforcing C's type system and checking the identifiers used.

This stage also introduces the concept of scopes, which track the accessible identifiers (names of variables and functions), tags (names of structs, unions and enums) and `typedef` aliases. After transformation, each translation unit is represented as a top-level scope, including definitions for functions and any static variables. Inside each function definition, there is a child scope extending the top-level scope, which adds definitions for the function's parameters, and then each compound statement recursively creates a child scope where variables are defined.

3.5.1 Type Checking

Each expression populates its own `type` field within its constructor and validates the types of its operands.

For example, the following snippet shows the class in the IR representing bit-shift expressions:

```
1  export class CShift {
2      readonly lvalue = false;
3      readonly type: CArithmetic;
4
5      constructor(readonly node: ParseNode, readonly lhs: CExpression,
6                  readonly rhs: CExpression, readonly dir: "left" | "right") {
7
8          this.type = integerPromotion(checks.asInteger(lhs.node, lhs.type));
9          checks.asInteger(rhs.node, rhs.type);
10     }
11 }
```

According to the C89 specification, both operands of the shift operator must be integers, and this is checked by the calls to the `checks.asInteger(node, type)` helper function. If either `type` isn't integral, then the parse-node parameter `node` is used to throw an exception showing the location of the error in the source code. The shift expression has the same type as the left-hand side after integer promotion (which converts integral types smaller than an `int` to an `int`), achieved using another helper function, `integerPromotion`, to reduce duplication.

3.5.2 Declarations

In C, there is a difference between declaring and defining an identifier. Declarations are used to let the compiler know that an identifier has the specified type without providing its definition, for example, `int add(int, int);` or `extern double pi;`. Definitions provide function implementations or the values of declared variables, for example `int add(int a, int b) {return a + b;}` or `double pi = 3.14;`. Each identifier must have one definition but can have any number of declarations (a definition also declares the identifier).

Since forward references (using an identifier before it is declared) are not allowed, forward declarations are often used in C code to remove constraints on the order of function definitions and to allow mutually recursive functions.

This structure is implemented in the IR by having `CVarDeclaration`/`CFuncDeclaration` classes have a `definition` field, which is initially `null`. When a new declaration is found during transformation, it is added to the current scope. Then, when the definition is located, it replaces the declaration in the scope and the `definition` field on the declaration is set to the newly constructed `CVarDefinition`/`CFuncDefinition`. This allows the transformation to be performed in one pass whilst enabling forward declarations to point to the definition once it is located.

3.6 Linking

Implementing a linker was crucial for compiling more complex programs spread across multiple files. Conventional compilers work by individually compiling source files into machine code, which are then linked into a single executable.

By comparison, c2wasm performs linking before code generation, after transforming each file into IR. This allows the actual linking to be done using the existing foundations for forward declarations, which massively simplified the implementation of the linker. However, since there is no mechanism for storing the IR, all files in the program must be compiled at once. This would make integration with existing tooling (such as CMake), that compiles each file seperately before linking, difficult.

After each source file is transformed into IR, the `Linker` class analyses each scope to find the externally linkable identifiers. It ensures that the types of each identifier match before linking them using the `definition` field. Optionally, the linker can be passed one or more existing linkers, which are used as fallbacks for finding definitions. Finally, it populates lists of functions, imports and static variables to be emitted by the code-generation stage.

The standard library is linked first by itself and cached and is then provided as a fallback when linking programs. This saves on parsing and IR transformation if the compiler is invoked multiple times in the same process. The linker also provides unreachable-procedure analysis when using fallback definitions by recursively including the dependency lists on each IR definition. This ensures the needed parts of the standard library are included without including everything, which would bloat generated modules and slow down the following compilation steps.

3.7 WebAssembly Helpers

The WebAssembly helpers simplify the code-generation stage by abstracting away the details of binary WebAssembly modules. WebAssembly has closely related text and binary formats, but textual modules have to be converted before execution. Therefore, c2wasm produces binary modules to avoid including a converter library, which would increase the bundle size.

These helpers support all the features needed for the compiler, including imported (JavaScript called from WebAssembly) and exported (Wasm called from JS) functions, mutable and exported globals, as well as memory and data segments. All the encoding details are handled, including the LEB128 encoding (a variable-length code) used for integers and managing the indices used to refer to functions, types and locals. They can also be used standalone without the compiler and have independent tests.

WebAssembly instructions are each represented as functions taking any immediates (e.g. the local used by `local.get` or the memory offset for `i32.store`) that return another function. When the instruction is then put into an expression, the second function is called with a context object to give an instruction-instance object. This context includes the current types on the stack and the nesting depth, which is used to enable functionality such as call instructions checking the needed arguments and branches calculating the relative jump depth. Each instruction-instance object includes the instruction's encoded binary representation, information on parameters consumed from the stack, the type of result produced (if any), and any resources (memory, globals or locals) accessed or written to. This information is crucial for enabling later analysis and optimisation.

The following TypeScript shows how a factorial function could be defined:

```
let builder = new ModuleBuilder();
builder.function(/*parameters*/ [i64Type], /*result*/ [i64Type], (func) => [
    Instructions.local.get(func.args[0]),
    Instructions.i64.const(2n),
    Instructions.i64.lt_s(),
    Instructions.if(i64Type, [
        Instructions.i64.const(1n)
    ], [
        Instructions.local.get(func.args[0]),
        Instructions.local.get(func.args[0]),
        Instructions.i64.const(1n),
        Instructions.i64.sub(),
        Instructions.call(func.self),
        Instructions.i64.mul()
    ])
], /*export name*/ "factorial");
let module = builder.toBytes();
```

Running this TypeScript would produce the following binary:

`0061736d`	magic number	`0a`	CODE section
`01000000`	version number	`19`	section size
`01`	TYPE section	`01`	number of functions
`06`	section size	`17`	size of first function
`01`	number of types	`00`	zero additional locals
`60`	function	`20 00`	`local.get 0`
`01 7e`	one i64 parameter	`42 02`	`i64.const 2`
`01 7e`	one i64 result	`53`	`i64.lt_s`
`03`	FUNCTION section	`04 7e`	`if (i64 result)`
`02`	section size	`42 01`	`  i64.const 1`
`01`	number of functions	`05`	`else`
`00`	type 0	`20 00`	`  local.get 0`
`07`	EXPORT section	`20 00`	`  local.get 0`
`0d`	section size	`42 01`	`  i64.const 1`
`01`	number of exports	`7d`	`  i64.sub`
`09`	string length	`10 00`	`  call 0`
`666163746f7269616c`	`"factorial"`	`7e`	`  i64.mul`
`00`	export is a function	`0b`	`end`
`00`	function index	`0b`	`end`

This is equivalent to the following textual WebAssembly module, showing the clear mapping between the two formats.

```
(module
  (type (func (param i64) (result i64)))
  (func (export "factorial") (type 0)
    local.get 0
    i64.const 2
    i64.lt_s
    if (result i64)
      i64.const 1
    else
      local.get 0
      local.get 0
      i64.const 1
      i64.sub
      call 0
      i64.mul
    end))
```

3.8 Code Generation

The code-generation stage is responsible for transforming the linked IR representation (a list of IR functions and variables to emit) into a WebAssembly module, using the helper library.

3.8.1 Statements

WebAssembly has three types of structured instructions:

1. `if` instructions, which consume an `i32` value and execute the if body if it is non-zero or the optional else body otherwise. Branches targeting `if` instructions jump to after the end of the instruction.
2. `block` instructions always execute, and branches jump to after the instruction.
3. `loop` instructions always execute, and branches jump back to the start of the loop body.

The simplest C statements to generate are `if` statements, which are transformed by generating the condition followed by a WebAssembly `if` instruction with branches corresponding to the original C statement.

The three C loop types are each transformed into multiple structured instructions in pre-defined patterns to provide conditional execution, looping and correct semantics for `break` and `continue` statements:

`while (`*cond*`) {`
  body
`}`
	`loop // continue target`
	  cond
	  `if // break target`
	  body
	  `br 1 // jump to start of loop`
	  `end`
	`end`

`do {`
  body
`} while (`*cond*`)`
	`block // break target`
	  `loop // continue target`
	  body
	  cond
	  `br_if 1 // conditional jump`
	  `end`
	`end`

`for (`*init*`;` *test*`;` *update*`) {`
  body
`}`
	init
	`loop`
	  test
	  `if // break target`
	  `block // continue target`
	  body
	  `end`
	  update
	  `br 1 // jump to start of loop`
	  `end`
	`end`

Switch statements are transformed by first evaluating the target expression and storing it in a temporary local. Recursive WebAssembly `block` instructions are then generated for each `case` body in order, for correct fall-through, wrapped in a top-level `block` for breaking out of the switch statement. The deepest WebAssembly block is populated with comparisons between each case value and the temporary local followed by corresponding conditional branches. These are followed by an unconditional branch to the `default` case body (or the top-level `block` if there is no default branch). A small example is shown below:

```
switch (expr) {
  case 1:
    body1
  case 2:
  case 3:
    body2
  default:
    body3
  case 4:
    body4
}
```

```
temp = expr
block // break target
  block
    block
      block
        block
          temp == 1
          br_if 0 // branch to body1
          temp == 2
          br_if 1 // branch to body2
          temp == 3
          br_if 1 // branch to body2
          temp == 4
          br_if 3 // branch to body4
          br 2 // default to body3
        end
        body1
      end
      body2
    end
    body3
  end
  body4
end
```

Generating `break` and `continue` statements is simple as the IR transformation works out which loop/switch statement they are targeting, and when generating each loop/switch statement, the correct branch depths are added to their IR representation.

I had planned to not support C `goto` statements, as they enable arbitrary control flow, which is unsupported in WebAssembly. Existing compilers work around this by transforming functions into control-flow graphs, which are then transformed into WebAssembly-compatible structured control flow (see Emscripten's Relooper algorithm[27]). Unfortunately, this would have added substantial complexity and be out of my project's scope.

However, to increase compatibility with real-world C code, limited `goto` support was added. It only supports branching outwards, like native WebAssembly, to avoid needing a complex control-flow transformation. This means it supports jumping out of nested loops but not into the body of an `if` statement, for example. Each compound statement is also restricted to at most one labelled statement, which simplifies code generation dramatically whilst not affecting most use cases.

All the statements before the labelled statement are placed in one WebAssembly `block`, allowing jumps forward to the label. The remaining statements are then placed within a `loop` instruction, enabling backwards jumps to the label.

```
{
  [body1...]
  label:
  statement
  [body2...]
}
```

```
block
  [body1...]
end
loop
  statement
  [body2...]
end
```

3.8.2 Value Types

Every WebAssembly value has one of four types: 32-bit integer (`i32`), 64-bit integer (`i64`), 32-bit floating point (`f32`) or 64-bit floating point (`f64`). The integer types can be used for signed or unsigned numbers and instructions either work on both two's complement and unsigned numbers (e.g. `i32.add`) or there are separate instructions (e.g. `i32.div_s` and `i32.div_u`). Values can be stored into and loaded from the untyped memory, and smaller integers can also be stored and loaded (e.g. `i32.store8`).

The C standard only defines minimum sizes for each integer type, and the actual size is implementation dependent. The following table shows the size of each data type in c2wasm and the value type used:

C arithmetic type	Minimum size (bits)	c2wasm size (bits)	Wasm value type
`char`	8	8	`i32`
`short`	16	16	`i32`
`int`	16	32	`i32`
`long`	32	64	`i64`
`long long`	64	64	`i64`
`float`	N/A	32	`f32`
`double`	N/A	64	`f64`

As the `int` type is meant to be the most efficient integer type, I made it 32-bits, like most modern compilers. Furthermore, `long` values are 64 bits to enable pure C89 code to use `i64` values, as `long long` was added by C99.

Finally, the current WebAssembly specification supports up to 4GiB of memory, indexed using `i32` pointers, so pointers in c2wasm are 32-bit.

3.8.3 Expressions

Due to WebAssembly's stack-based nature, expression generation is similar to creating postfix/Reverse Polish notation, i.e. push the operands to the stack, then perform the operation. My implementation has a main `expressionGeneration` function, which delegates to a type-specific function. All the functions also take a context object and a flag to indicate whether the expression's result should be discarded. When this flag is set, only code that may have side effects is generated, and each function must drop any created values from the stack.

Below is a simple example of this generation code for `==` and `!=` expressions. If the result is discarded, the left and right-hand subexpressions are recursively generated with the discard flag set. Otherwise, it uses the `subExpr(..., e.commonType)` helper function to generate each subexpression and insert any needed type-conversion code, and then the `gInstr(type, instrName)` helper to insert the correct comparison instruction (e.g. `i32.eq`, `f64.ne`).

```
function equality(ctx: WFnGenerator, e: CEquality, discard: boolean) {
    if (discard) {
        return [ // generate any side effects
            ...expressionGeneration(ctx, e.lhs, true),
            ...expressionGeneration(ctx, e.rhs, true)];
    }
    return [
        ...subExpr(ctx, e.lhs, e.commonType),
        ...subExpr(ctx, e.rhs, e.commonType),
        gInstr(valueType(e.commonType), e.op === "==" ? "eq" : "ne")];
}
```

3.8.4 Storage

In c2wasm, every variable and memory location has an associated `StorageLocation` object, which defines how it is loaded and stored. When generating each scope, these locations are initialised (for example, allocating static memory space) and tagged onto the IR representation. The expression-generation functions then use a set of storage helper functions to generate reads, writes or updates to locations. This abstracts the location's type and the needed instructions, which can be complex and require duplicating or reordering stack values.

There are four storage-location types:

1. *Local:*
   Non-static arithmetic or pointer variables inside functions are stored using WebAssembly function locals unless their address is taken (as WebAssembly prevents arbitrary access). The `StorageLocation` object stores the type as `"local"` and the allocated local's index.

2. *Shadow:*
   Used to store local `struct`/`union` variables as well as variables that have their addresses taken. These values are stored on a second 'shadow' stack in WebAssembly memory (see section 3.8.5). The `StorageLocation` instance stores the allocated offset from the shadow-stack pointer.

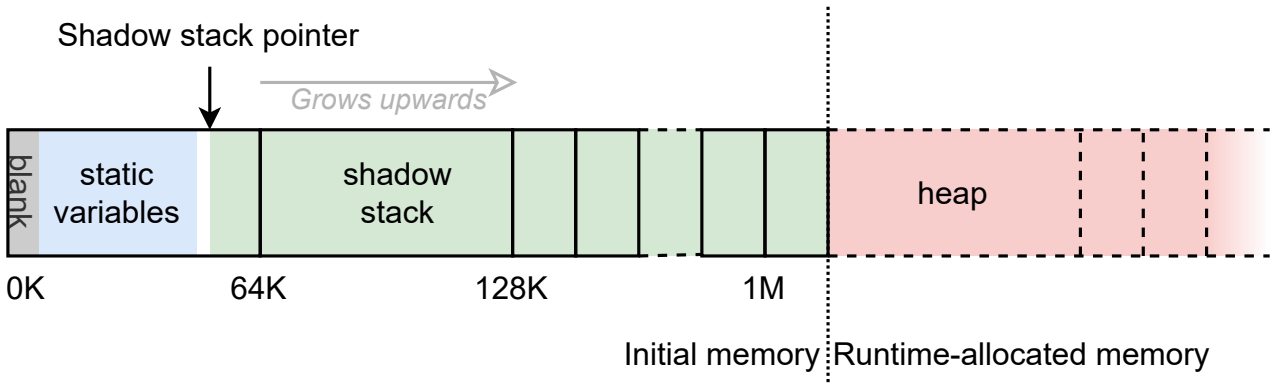
3. *Static:*
   Used to store `static` variables, which are stored at preallocated addresses. The address is stored as part of the `StorageLocation` instance.

4. *Pointer:*
   Used when the location is an arbitrary pointer that is on top of the WebAssembly stack. This type is never associated with variables but is used for accessing dereferenced pointers, for example.

3.8.5 Memory Layout

In WebAssembly, memory is exposed as a contiguous array of untyped bytes and measured in 64 KiB pages. Modules have to declare a minimum initial memory size (in pages) to enable memory support. At runtime, the current number of pages can be queried using the `memory.size` instruction, and the memory can be dynamically grown using `memory.grow`.

The following diagram shows how memory is used in modules generated by c2wasm. Each rectangle represents one page.



- At the start of the memory is a small blank region preventing static variables from being allocated at address zero, which would equal the `NULL` pointer.
- Following this is a region containing the static variables, each at preallocated addresses. This region is dynamically sized at compile time to fit the static values and any padding needed for their natural alignment.
- The next region is used for the shadow stack and begins following some padding. The shadow-stack pointer is stored as a WebAssembly global, with the initial value being the lowest address in the region. This means function calls increase the stack pointer, the opposite of the historical convention where the stack grows down from the top of memory. However, as the memory space is dynamically resizable, the top address is not fixed, so I decided to use a fixed size for the shadow stack. This is achieved by taking the start address, adding on 1 MiB (by default), and then rounding up to set the module's minimum memory pages.

- The final region is heap used by the `malloc` implementation (see section 3.9.2). The entire region is allocated using `memory.grow` at runtime, unlike the previous regions stored in the initial memory. This means that programs using `malloc` can use all the remaining memory space beyond the minimum memory as a heap.

However, there are potential issues, which are difficult to fix due to WebAssembly's constraints:

1. The shadow-stack pointer can grow outside the shadow stack region and overwrite the heap or crash. This can be prevented by ensuring the shadow stack is large enough, and the default of 1 MiB was chosen as it is more than enough for all the programs tested.
2. The `null` pointer can be dereferenced and the initially blank region can be written to. This could be fixed by using a non-zero internal representation (allowed by the C89 specification) that is outside the allocated address space. However, modules can allocate the entire 32-bit address space (implementation allowing), so no `i32` representation would always be safe.

Alternatively, these issues could be fixed by inserting checks into the generated code, but this would substantially degrade performance. In traditional applications, these problems are fixed using guard pages that trap on attempted access, but this is not possible from WebAssembly.

3.8.6 Calling Convention

WebAssembly function definitions include a list of argument types, and these arguments are consumed from the stack when calling a function. However, this can only be used to pass a predefined list of WebAssembly values. As `struct`/`union` values are not representable as WebAssembly values, they are passed as a pointer and then the callee copies the value into its shadow stack region. Calls to variadic functions (ones accepting variable numbers of arguments, e.g. `printf`) store the additional arguments on top of their shadow stack region, which is just under the callee's region, enabling access via negative offsets. The caller is responsible for increasing the shadow-stack pointer prior to the `call` instruction and restoring it after.

The current WebAssembly specification allows functions to return at most one WebAssembly value and is used by functions returning arithmetic/pointer values. Larger `struct`/`union` values are returned by adding an additional argument into the function definition, which the caller uses to pass a pointer to space allocated on the shadow stack. The callee then copies its result to the given address before returning.

Support for function pointers was another feature that I had planned to exclude but was required for many programs, including the `printf` implementation. Indirect WebAssembly calls are parametrised by the function's type and consume an index from the stack at runtime, which is then looked up in a table of function indexes in the module to select the function to call. Adding support was easier than expected as the C function pointer's type information can be transformed to give the parametrised WebAssembly type, and the function table is populated by tracking which functions have their address taken during IR transformation. Finally, when generating an indirect function call, evaluating the function pointer is the final step before the function-call instruction (as the index has to be on top of the stack), and the normal `call` instruction is replaced with `call_indirect`.

3.8.7 Internal Functions

Another important feature of the code-generation stage is the support for "internal" functions. These functions are parsed and transformed as normal, but are passed to a separate code-generation function and are not generated like real function calls. They are declared in a special scope that is used as the parent scope for all translation units. There are five internal functions for inline WebAssembly, and two for accessing variadic arguments.

Allowing arbitrary WebAssembly to be embedded inside C code is crucial for supporting instructions that are never generated from standard C syntax. This is used by the standard library to allocate memory at runtime, for example, but could be used by programs written for `c2wasm` to access future instructions or for hand-crafted WebAssembly. Four of the functions are used for instructions resulting in each of the value types (`__wasm_[i32/i64/f32/f64]__`) and the final function for instructions with no result (`__wasm__`). Each function takes a constant integer argument indicating how many of the variadic arguments are values that should be pushed onto the stack before the arbitrary code, and then the remaining arguments must be constants representing WebAssembly instructions. However, as the WebAssembly helpers support writing but not reading instructions, these instructions cannot be analysed by the optimisation passes. This limits the instruction support to instructions not affecting control flow or accessing locals (which may be remapped). Below is an example of inlining a `memory.grow` instruction for requesting more memory at runtime and the generated WebAssembly:

```
int requestMemory(int wasmPages) {
  int result = __wasm_i32__(
      /* # of args  */ 1,
      /* argument 1 */ wasmPages,
      /* Wasm bytes */ 0x40, 0x00);
  return result != 0; // success = 0
}
```

```
(func (param i32) (result i32)
  local.get 0
  memory.grow // 0x40 0x00
  i32.const 0
  i32.ne)
```

The other internal functions are used to implement the `<stdarg.h>` variadic argument macros (see section 3.8.6). The first, `__wasm_ssp__()`, returns the current shadow-stack pointer, and is used by `va_start` as these arguments are stored right below the callee's pointer.

```
#define va_start(ap, parmN) ( ap = (char*) __wasm_ssp__() )
```

Accessing the variadic arguments is achieved by decreasing the pointer 8 bytes at a time, and loading from it. This is performed using the final function, `__wasm_rload__(ptr)`, which acts like a normal pointer dereference unless the pointer represents a `struct`/`union`. If it does, an extra memory load is inserted to account for the value being passed as a pointer.

```
#define va_arg(ap, type) ( ap -= 8, *((type*) __wasm_rload__((type*) ap)) )
```

3.8.8 Function Import and Exports

The final feature necessary for the standard library is support for declaring WebAssembly function imports. These functions have to be supplied when running the module and are implemented in the host environment's language (e.g. JavaScript in the browser). This was implemented by adding an `import` function-declaration modifier, which causes an import to be generated matching the declaration. Additionally, the compiler enforces that these declarations do not have any C definitions that would conflict with the import.

The JavaScript code below shows usage of the c2wasm API to compile a C code snippet, including the declaration of an imported `random` function (line 2). When this module is instantiated, a compatible import has to be provided, or an error is raised, which is achieved by providing the JavaScript `Math.random` function (line 9). Finally, c2wasm exports all externally linkable (non-`static`) functions that aren't from the standard library in generated modules. This allows the `uniform` function, defined using C, to be called from JavaScript (line 13).

```
1  const module = await c2wasm.compile(/* C source */`
2    import double random();
3
4    double uniform(double min, double max) {
5      return min + (max - min) * random();
6    }
7  `).execute(/* Imports */{
8    c2wasm: {
9      random: Math.random
10   }
11 })
12
13 module.uniform(10, 20);
14 > 15.178807772507882
```

3.9 C Standard Library

Implementing the C standard library was tricky due to c2wasm's limited support for more modern C features or non-standard compiler extensions used by most existing implementations. Many also rely on build tools such as `autoconf`, `make` or `cmake`, none of which are supported by c2wasm, which only has a JavaScript/TypeScript interface. Furthermore, I wanted to avoid linking copyleft-licensed code into the modules generated by my compiler. These limitations prevented using any existing complete standard library that I could find, so c2wasm's standard library contains a mix of functions from existing libraries and some of my own implementations. The existing libraries used were mpaland/printf[28] (MIT Licence), musl[29] (MIT Licence) and AVR Libc[30] (Modified BSD Licence).

Appendix B details header-file support and where the existing libraries were used. Out of the 15 original C89 header files, 7 are fully supported, 4 partially, and 4 are unsupported. Some C99 headers including `stdint.h` and `stdbool.h` are also supported. This goes far beyond the plan in my project proposal to include only essential functions such as `malloc` and `free`.

The rest of this section describes some of my implementations in the standard library.

3.9.1 IO

IO was challenging to implement as it requires interfacing outside the WebAssembly module. Most C libraries target POSIX syscalls, which isn't possible from WebAssembly without the host environment exposing the syscalls through an interface such as the upcoming WebAssembly System Interface (WASI)[31]. I created two custom implementations for c2wasm, one that only supports a single text output and another that emulates a basic file system:

Simple Text Output

Each character is written serially by calling an imported `__put_char(char)` function, similar to the `putchar` function exposed in `<stdio.h>`. Since this is the only import, instantiating a compiled module requires providing only a simple character-output function.

File-system Emulation

Emulating a file system was needed to support benchmarks and applications that read in one file and then produce another. When compiling a module, this implementation is used if the preprocessor macro `FILES` is defined. It implements basic support for all the `<stdio.h>` functions, but requires a more complex wrapper at runtime, providing the following imports:

```
import int  __put_char(int handle, int c);
import int  __get_char(int handle);
import int  __set_pos(int handle, long pos);
import long __get_pos(int handle);
import long __get_len(int handle);
import int  __exists();
import int  __move();
import int  __get_fhandle();
```

The runtime wrapper stores a map of file names to file handles and a map of file handles to file objects. Each file has a unique file handle that never changes, and the wrapper has no concept of open or closed files. The file objects include a byte array storing the file contents and the current position within the file.

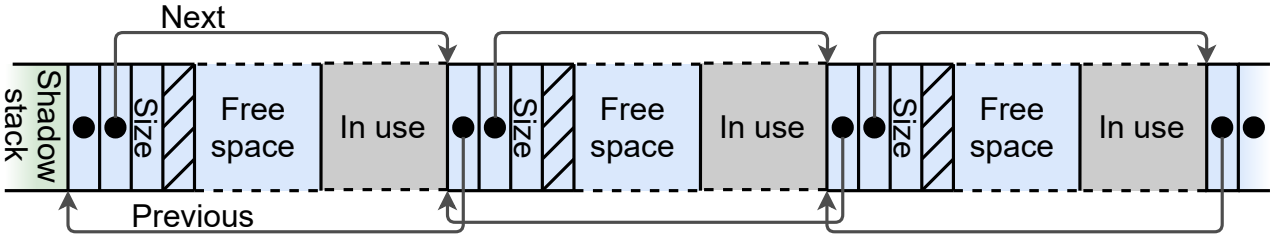
Opening a file from C first involves transferring the provided file name to the wrapper, which is achieved by writing each character sequentially to a special file handle. The `__get_fhandle()` import is then called, which looks up the provided file name and returns a file handle that is then stored within the heap-allocated FILE struct. Finally, the file position is reset depending on the provided access mode (which is otherwise ignored).

The `__exists()` and `__move()` imports are used similarly with file names written to the special file handle first. However, `__move()` takes two file names separated by a zero byte, and if the second name is blank, the target file is deleted instead.

Appendix E demonstrates how the runtime wrapper is used with the c2wasm API.

3.9.2 Memory Allocation

My implementation of `malloc` was heavily inspired by Embedded Artistry's first-fit free-list allocator design for embedded systems[32], but I rewrote it to not depend on their linked-list library and with modifications for WebAssembly. Each unallocated region has a header `struct` storing forwards and backwards pointers to neighbouring regions and the region's size, as shown in the following diagram:



Allocating memory is achieved by walking down the linked list until a large enough region is found. The free space in the region is then allocated to the caller, and the list node is removed from the linked list. If the region size is above a set threshold over the requested size, then the region is split, and a new linked-list node is constructed after the allocation. If no region is large enough or this is the first allocation, runtime-allocated memory pages are requested using inline WebAssembly. A new node is then built at the start of the new pages and linked into the list before repeating the algorithm.

Memory is freed by walking down the linked list and inserting the region in the correct position to keep the list in order. Neighbouring unallocated regions are then merged to avoid fragmentation.

3.10 Optimisations

The final step in the compilation process is the optimisation stage, which is unusual as most compilers perform the majority of optimisations on various intermediate representations before code generation. However, adding optimisations was predominately an extension to the project, and adding extra IR stages and substantially modifying the code-generation stage was not feasible given the time constraints of the project. I instead used this time to focus on implementing as many useful optimisations as possible on the generated WebAssembly.

See the preparation chapter, section 2.4, for descriptions of all the implemented optimisations.

3.10.1 Peephole Optimisations

Peephole optimisations scan through the generated instructions and perform predefined replacements. They are mainly used to clean up the generated code in a separate pass, simplifying the code-generation stage.

The following peephole optimisations are implemented (where T stands for any value type):

- `local.set [idx]; local.get [idx]` → `local.tee [idx]`
- `i32.const [a]; i32.add; T.load(offset=[b])` → `T.load(offset=[a + b])`
- `i32.const [a]; i32.add; i32.const [b]; i32.const` → `i32.const [a + b]; i32.add`
- `i32.const [a]; i32.const [b]; i32.[op]` → `i32.const [a op b]`
- `i32.const [a]; i32.eqz` → `i32.const [a != 0]`
- `T.const [a]; T.const [b]; T.add` → `T.const [a + b]`
- `T.const [a]; T.const [b]; T.mul` → `T.const [a * b]`
- Removing `T.const 0; T.add`
- Replacing `block` and `loop` instructions that are never branched to with their body.
- Replacing `if` instructions with constant conditions.

These replacements focus on `i32` operations as these are used for pointer calculations, which are generated naively. However, more patterns for other value types could easily be added. Care has to be taken when replacing a structured instruction with its body to ensure outward branches have their relative jump depth modified accordingly.

3.10.2 Dead-code Elimination

The code-generation stage rarely generates dead code due to the implementation of the `discard` flag (see section 3.8.3). However, other optimisations (e.g. copy propagation, section 3.10.5) can produce dead code, which is removed by this pass.

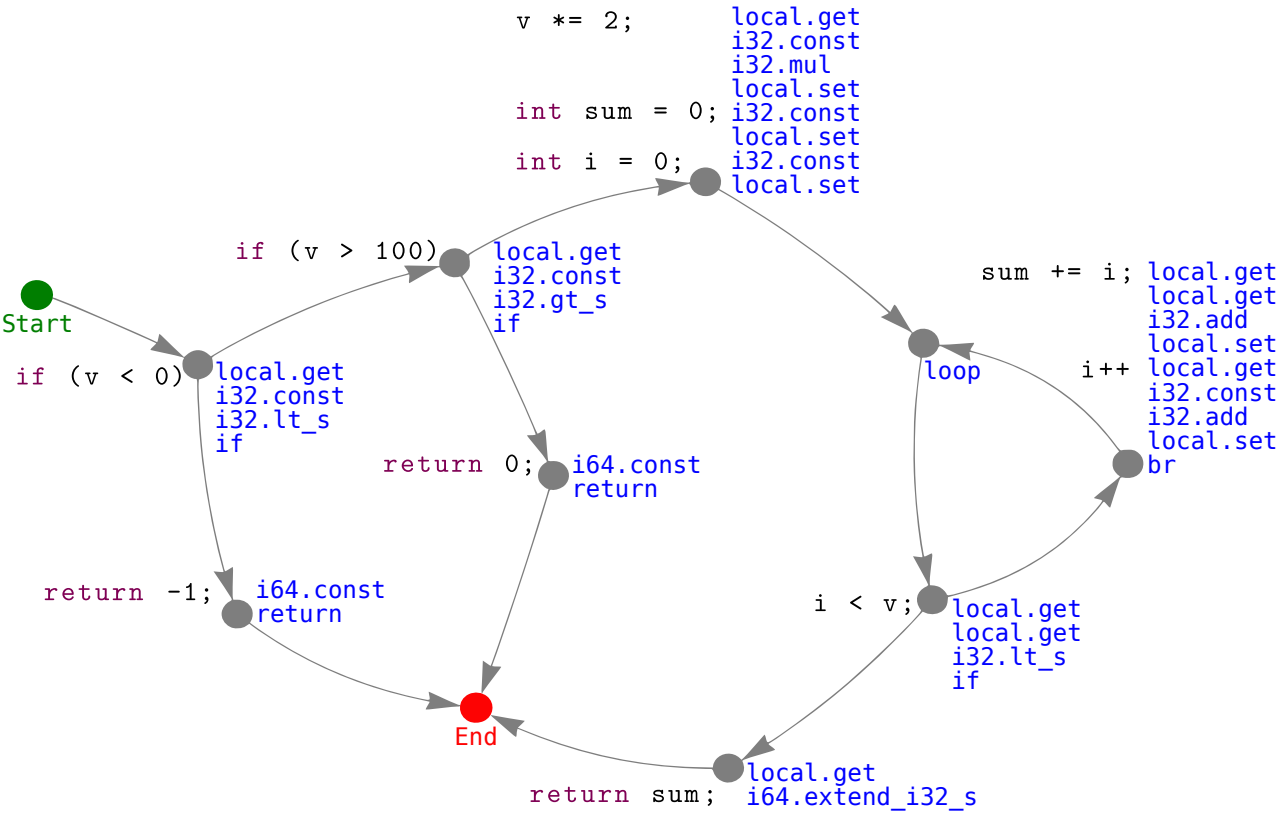
The implementation works by first marking all instructions with side effects as needed. It then looks at the stack items consumed by these instructions and recursively marks the instructions that produce these values. Once every dependency is marked as needed, unneeded instructions are removed. If an unneeded instruction consumes the result produced by a needed instruction, then `drop` instructions are inserted to prevent leaving extra values on the stack (which is an error in WebAssembly).

3.10.3 Control-flow Graphs

Control-flow graphs (CFGs) represent all possible execution paths and are needed for many optimisations. Most compilers use IR representations that natively encode control flow, but since c2wasm performs optimisations on WebAssembly instructions, the CFG has to be built separately. Each instruction is mapped onto a node object and is iteratively linked to the instructions that could follow. This is usually the following instruction but is different for structured and branch instructions. Finally, backwards links are added to the nodes.

To aid in debugging optimisations, I created a demo to visualise CFGs (see Appendix D). Below is the output for the simple `cfgDemo` function (right), manually edited to add the corresponding C code.

```
long cfgDemo(int v) {
    if (v < 0) return -1;
    if (v > 100) return 0;
    v *= 2;
    int sum = 0;
    for (int i = 0; i < v; i++)
        sum += i;
    return sum;
}
```



The visualisation combines neighbouring nodes to simplify the graph, similar to the creation of basic blocks used to speed up analyses. However, my implementations do not use basic blocks. Instead, most filter the CFG to only include relevant instructions (e.g. `local.*` instructions for live-variable analysis) by removing the nodes that would usually be combined.

3.10.4 Data-flow Equations

Data-flow equations can be expressed in the standard form shown below:

$$in_x = [\cup/\cap]_{y \in [pred/succ]_x}(out_y) \qquad out_x = f_x(in_x)$$

Each node x has an in-state in_x and out-state out_x. Depending on the equation, the in-state is either the union or intersection of the out-states of its predecessors (forwards data-flow equations) or successors (backwards equations). The out-state is calculated using the equation's transfer function f, which depends on the in-state and the node in question.

The fixed point can then be found by recomputing each node's state until there are no more changes. This is implemented in a parametrised helper function to simplify the implementations of data-flow based optimisations. The nodes to recompute are stored in a queue, and if a node's state changes after recomputation, its predecessors/successors are added back into the queue. These recomputations involve large numbers of set operations, which became the limiting factor when representing states as JavaScript sets.

My final implementation represents each state as a JavaScript BigInt, an arbitrary length integer used as a bitstring. This substantially increases performance by replacing set operations with simpler bitwise operations, such as bitwise OR for unioning sets. However, it requires additional analysis code to assign each value a unique bit and interpret the results.

Simplified TypeScript for the helper function is shown below:

```
function framework(cfg: ControlFlowGraph,
      direction: "forwards" | "backwards",
      meetOperation: "union" | "intersection",
      transferFunction: (f: Flow, x: bigint) => bigint): Map<Flow, bigint> {
  var queue = new Queue<Flow>();
  var result = new Map<Flow, bigint>();

  for (var node of cfg) { queue.add(node); result.set(node, 0n); }

  while (queue.size() > 0) {
    var node = queue.dequeue(); // remove next node to recompute
    var value = meetOperation == "union" ? 0n : -1n;

    for (var n of (direction == "forwards" ? node.prev : node.next)) {
      if (meetOperation === "union") value |= result.get(n);
      else /* intersection */        value &= result.get(n);
    }

    value = transferFunction(node, value);
    if (value == result.get(node)) continue;
    result.set(node, value); // value didn't match

    for (var n of (direction == "forwards" ? node.next : node.prev))
      queue.add(n); // recompute dependents
  }
  return result;
}
```

3.10.5 Copy Propagation *(see section 2.4.1)*

Copy propagation uses reaching-definition analysis to find safe `local.get` instructions to replace. After building the control-flow graph and simplifying it to `local.*` instructions, the next step is to find every local definition (`local.set` and `local.tee` instructions). A definition-use chain object is constructed for each one, containing the local's index, a uniquely allocated bit, and lists to store possible and definite uses. Additionally, an array of bitmasks is precomputed, where each entry represents all the definitions for a particular local. The unique bits and bitmasks are then used by the transfer function passed to the equation helper:

```
framework(cfg, "forwards", "union", /* transferFn */ (node, x) => {
    if (node represents a definition) {
        // unset any existing definitions for the local set by the node
        x &= ~localMasks[node.local];
        // add the definition's unique bit
        x |= node.uniqueBit;
    }
    return x;
});
```

The results then represent all reachable definitions for all locals at each node. For each `local.get` node, the bitwise AND of its result and the accessed local's bitmask is computed to give a list of all definitions that could be accessed when running the instruction. This node is then added to the possible uses of all the possible definitions. If there is only one possible definition, it is also added to that definition's definite uses.

After the analysis, the next step is iterating through each definition, performing any possible optimisations. If the definition stores a constant value into the local, then any definite uses can be replaced with the constant (constant propagation). Otherwise, if the local is set to the value of another local that isn't modified before a definite use, the use can be replaced with the other local (copy propagation). Finally, definitions that were never used or had all their usages replaced are removed. This is achieved by replacing `local.tee` instructions with `nop` instructions and `local.set` with `drop` instructions, which are subsequently cleaned up by a dead-code elimination pass.

3.10.6 Local Reallocation *(see section 2.4.1)*

By default, c2wasm reuses locals based on the scoping of C variables. However, enabling local reallocation disables this behaviour, giving each variable (and temporary) a unique local. This optimisation pass then reallocates the WebAssembly locals, often resulting in fewer being used.

LVA (live-variable analysis) is simple to perform on WebAssembly locals, as they have unique indices, making the preallocation of unique bits unnecessary. The clash graph is then constructed by iterating over the results and adding edges between simultaneously live locals. Finding locals to combine is equivalent to graph colouring, which is NP-complete, so I implemented the heuristic suggested in the Optimising Compilers Part II course.

The first step is to copy the clash graph and continuously remove the node with the fewest edges onto a (LIFO) stack. A new list of locals is constructed, and each node is sequentially removed from the stack and checked against this list. The node is allocated to the first new

local with the correct value type and no existing node mappings that clash with this node. If no such local exists, an additional local is allocated instead and added to the list.

Finally, all the `local` instructions need to be remapped accordingly, which is achieved using a peephole pass that replaces such instructions based on a mapping between old and new indices.

Live-range Splitting

Live-range splitting tries to split locals that have been reused unnecessarily, the opposite of local reallocation. However, when enabled, it runs before the reallocation pass, splitting any reused variables, which may allow for more efficient reallocation.

The implementation is similar to reaching definitions but does not use the data-flow equation solver. Each definition node starts out defining its own "local range", but these are merged per local variable where CFG nodes join. This is equivalent to partitioning a reaching-definition graph but implemented in a single pass. If the process ends with more local ranges than locals, each local range is remapped to its own local.

3.10.7 Partial-redundancy Elimination *(see section 2.4.1)*

PRE removes computations that are redundant on some paths through the control-flow graph, and the implementation was based on "*A simple algorithm for partial redundancy elimination*" by V.K. Paleri[25].

The first step is finding all the subexpressions, which is achieved by finding all the instruction sequences longer than two instructions that leave one value on the stack. Additionally, the instructions cannot consume values generated before the first instruction or have side effects. The subexpressions are then deduplicated and converted into objects. Each one stores all the positions the expression occurs at, any accessed resources and a unique bit for the framework.

Next, each node's local properties (transparency, availability and anticipability) are computed as bitmasks. The set of equations a node is transparent to is equivalent to the expressions that don't use any resources overlapping with the node's side effects. The set of locally available expressions at each node is the set of subexpressions ending on that node and is the same as the locally anticipable expressions, as my implementation does not use basic blocks.

The following step is to perform the four data-flow analyses using the solver function to establish global properties. The first and second analyses calculate global availability and anticipability using local availability and anticipability, respectively, and node transparency. Next, safe locations to insert computations for each expression are computed by unioning global availability and anticipability at each node. The third and fourth analyses then calculate safe partial availability and safe partial anticipability, which require that points along the partial availability or anticipability paths are also safe locations to insert computations.

The final analysis step is to use the results of the analyses by iterating over every node and evaluating the formulas given in the paper to find the insertion nodes and edges as well as the replacement points. Performing the optimisation then requires inserting and replacing instructions throughout the WebAssembly.

3.10.8 Function Inlining *(see section 2.4.2)*

Function inlining is an interprocedural optimisation, meaning it runs after all the functions are defined and individually optimised. It first analyses all the usages of `call` instructions to build a map of function calls. A score-based heuristic is then evaluated for each function:

```
score = numInstructions + 5 * min(numArgs - 1, 0) + 5 * numLocals;
if (C function marked with inline hint) score -= 20;
```

If this score is less than or equal to 8, then the function is always inlined. Otherwise, if the score is less than or equal to 16, and the function is not exported, or in the indirect call table, it is inlined if it has fewer than 4 uses and removed.

Inlining is performed by first allocating new locals for the inlined function's arguments and locals. The `call` instruction is then replaced by successive `local.set` instructions to store the function's arguments, followed by a `block` instruction resulting in the same WebAssembly type as the original function. The function body is then copied inside, the locals are remapped, and any return statements are replaced with branches targeting the `block` instruction. Finally, the function is individually optimised again.

The scoring function was manually tuned. Functions are penalised for having more than one argument due to the overhead of the `local.set` instructions, which are unlikely to all be optimised out. Additionally, inlining functions shouldn't drastically increase the number of locals needed by a function, so non-argument locals are also penalised.

3.11 Summary

In this chapter, I have described the implementation and structure of my project, c2wasm, and the challenges faced. Every step in the compilation process has been discussed, and the code-generation and optimisation stages have been described in detail.

4 Evaluation

This chapter evaluates my project, c2wasm, using the following five metrics:

- The runtime performance and size of the produced WebAssembly modules.
- The size of the compiler (as a metric for its suitability for the web).
- C language support and correctness.

Finally, I evaluate my project's achievements and if the original success criterion was met.

4.1 Performance

The benchmarks were all run automatically, using a custom framework which handled compiling programs using c2wasm, Emscripten and GCC, as well as executing each benchmark in each configuration 100 times. See Appendix G for more details.

The following three benchmarks were used:

1. A toy benchmark for demonstrating c2wasm's partial-redundancy elimination. See Appendix F for the source code and analysis of the generated WebAssembly.
2. CoreMark[33], a simple synthetic benchmark, used as an indication of overall performance.
3. cjpeg (from libjpeg 6b[34]), which compresses an image into a JPEG, as an example of a real workload.

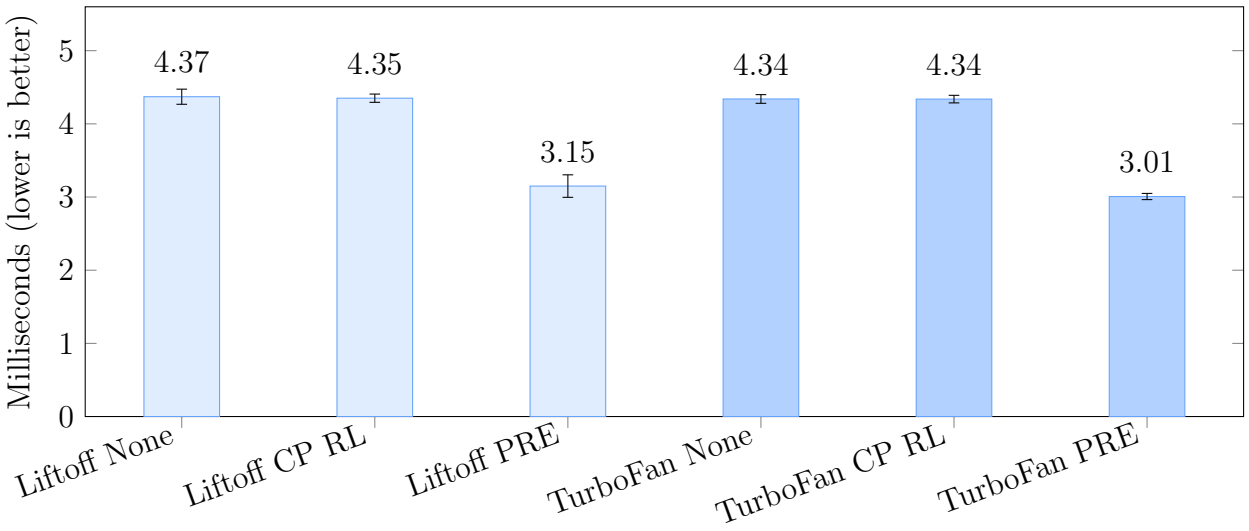
I tested c2wasm under the following six configurations, each enabling additional optimisations. For full details on the optimisation flags enabled in each configuration, see Appendix C.

1. None — No optimisations enabled.
2. ConstExpr — The code-generation stage tries to emit expressions as constants.
3. Peephole — Peephole optimisations (see section 3.10.1) enabled.
4. CP RL — Copy propagation (section 3.10.5) and reallocating locals (3.10.6) enabled.
5. PRE — Partial-redundancy elimination (section 3.10.7) enabled.
6. Inlined — Function inlining (section 3.10.8) enabled.

The bar charts are colour coded: blue for c2wasm, red for Emscripten (shortened to 'Emcc', the name of the frontend) and green for native. Additionally, the bars for modules run under TurboFan (see section 2.3.3) are darker. Error bars are used to show standard deviation, and the averages are labelled above each bar.

4.1.1 PRE Benchmark

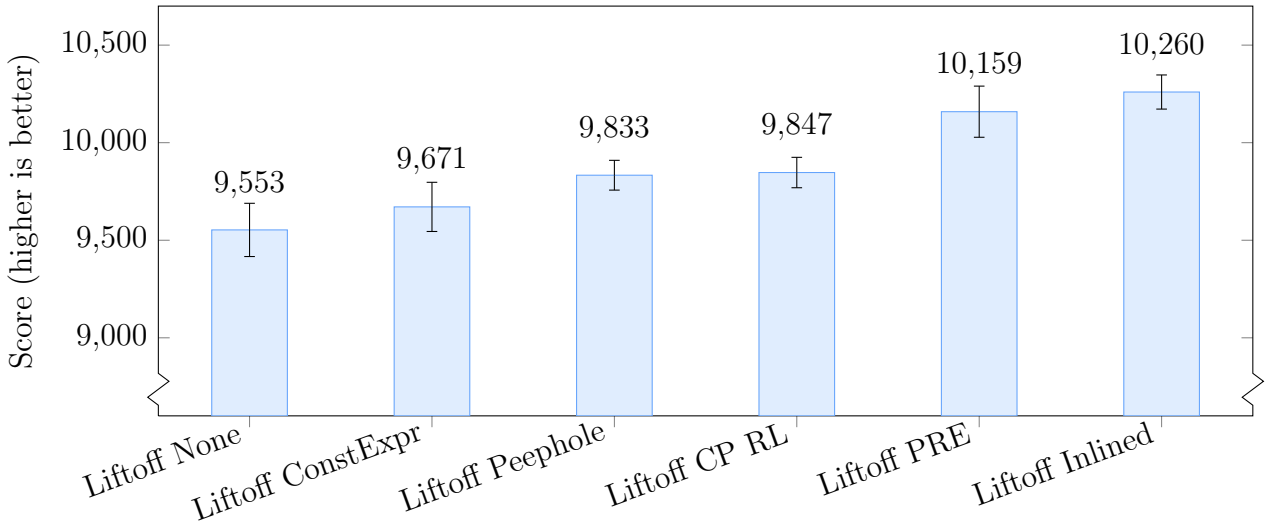
The PRE benchmark was designed to demonstrate partial-redundancy elimination and consists of a small function called in a loop. As detailed in Appendix F, the function contains one fully redundant calculation and one partially redundant calculation, which are both eliminated by c2wasm's PRE pass.



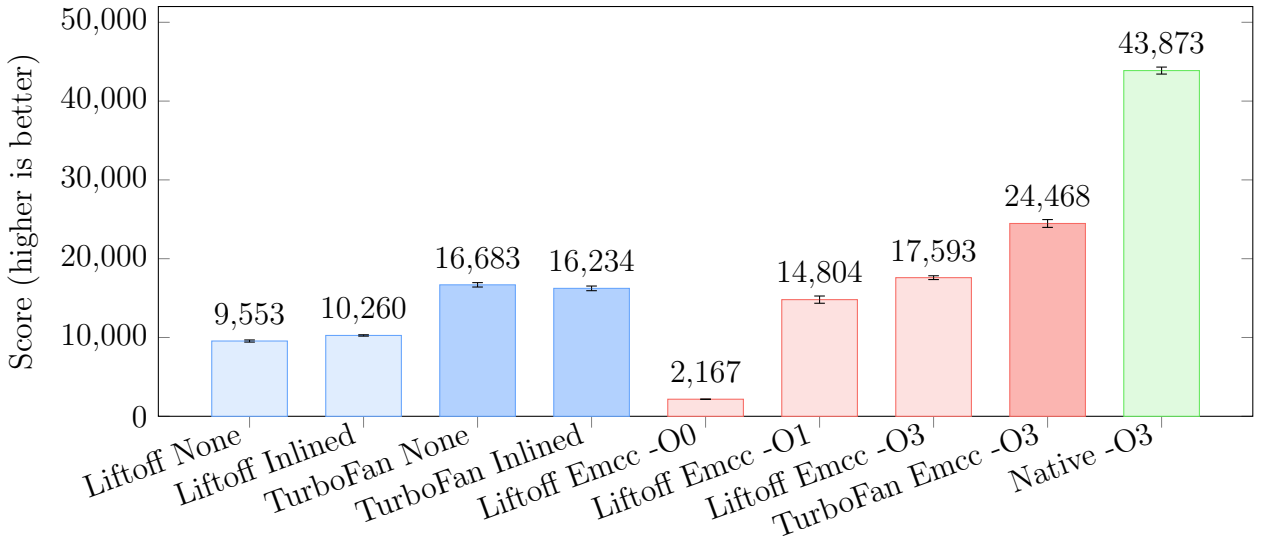
These results show that PRE improves performance by around 30%. Liftoff and TurboFan perform almost identically, which is surprising as Liftoff is a simple one-pass WebAssembly compiler, and TurboFan is a complex multi-pass compiler. This may indicate that the function was too short for any major optimisations and that TurboFan may not perform PRE.

4.1.2 CoreMark

CoreMark[33] is a simple synthetic benchmark made up of three main components: list processing, matrix manipulation and state-machine processing. Each component's result is also checked by calculating a cyclic redundancy check (CRC) value. This is repeated until 10 seconds have passed, and the score is the number of iterations a second.



Enabling partial-redundancy elimination resulted in the largest score increase of around 3%. This is significantly lower than the previous benchmark but shows PRE is useful for a more realistic workload. In total, the optimisations improve performance by around 7% under Liftoff, which is significant as shown by the error bars for 'none' and 'inlined' not overlapping.



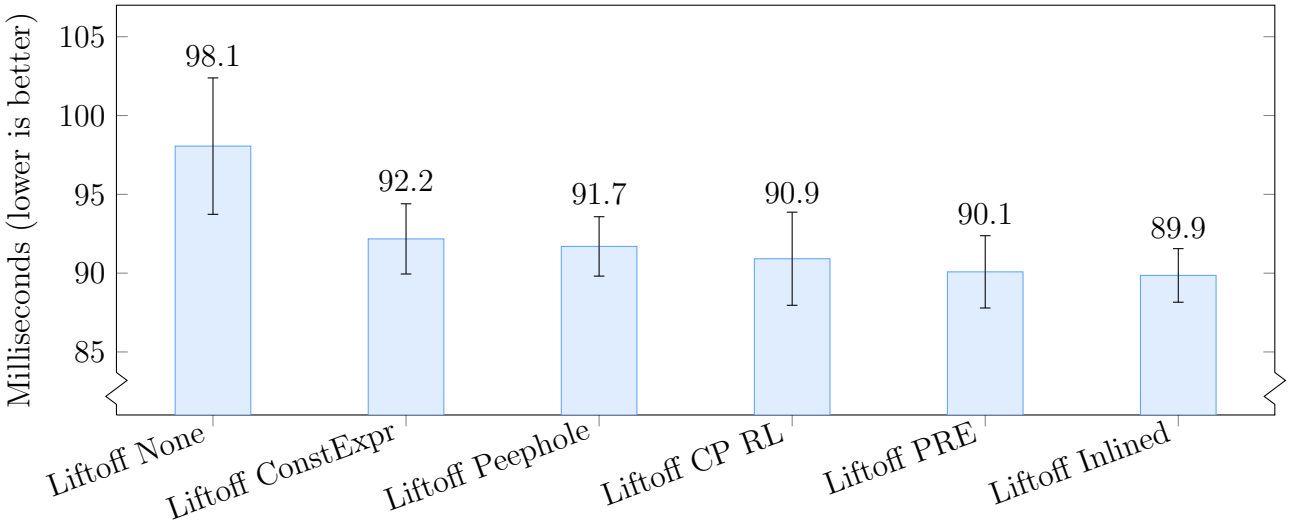
TurboFan offers substantially better performance overall, but c2wasm's optimisations slightly hurt performance, indicating that TurboFan can better optimise the simpler instructions produced by the code-generation stage.

My compiler's performance is between Emscripten's -O0 and -O1 optimisation levels. Emscripten's highest optimisation level (-O3) performs significantly better than c2wasm. However, its unoptimised output (-O0) performs appreciably worse than c2wasm's unoptimised output. This shows that Emscripten was built with optimisations in mind, unlike c2wasm where the optimisations were an extension to the existing code generation.

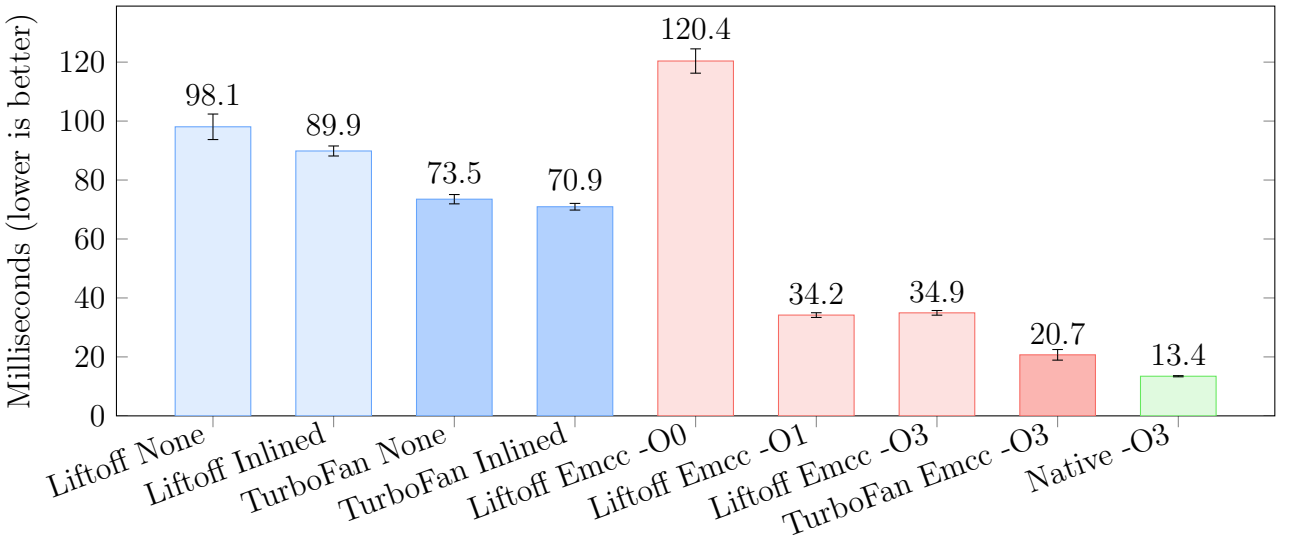
Finally, c2wasm under TurboFan achieves around 38% of native performance and Emscripten around 55%, showing that WebAssembly is limited to around half of native performance.

4.1.3 cjpeg

The final benchmark compresses a 1280x720 8-bit BMP image into a JPEG using cjpeg.



The only optimisation that offers a non-negligible improvement alone is emitting expressions as constants when possible. Unlike the previous benchmarks, this workload involves a lot of memory operations, which aren't affected by the majority of my optimisations. This is due to how difficult pointer analysis is, particularly when directly optimising WebAssembly. Additionally, this benchmark performs around 1MiB of IO and my simple IO implementation (see section 3.9.1) only transfers one byte at a time.



Using TurboFan only increases performance slightly, probably due to the IO overhead, but c2wasm's optimisations result in a small performance gain, unlike the previous benchmark. Similarly, c2wasm's performance is between Emscripten's -O0 and -O1 optimisation levels. However, there is a much larger difference between c2wasm's most optimised module and Emscripten's.

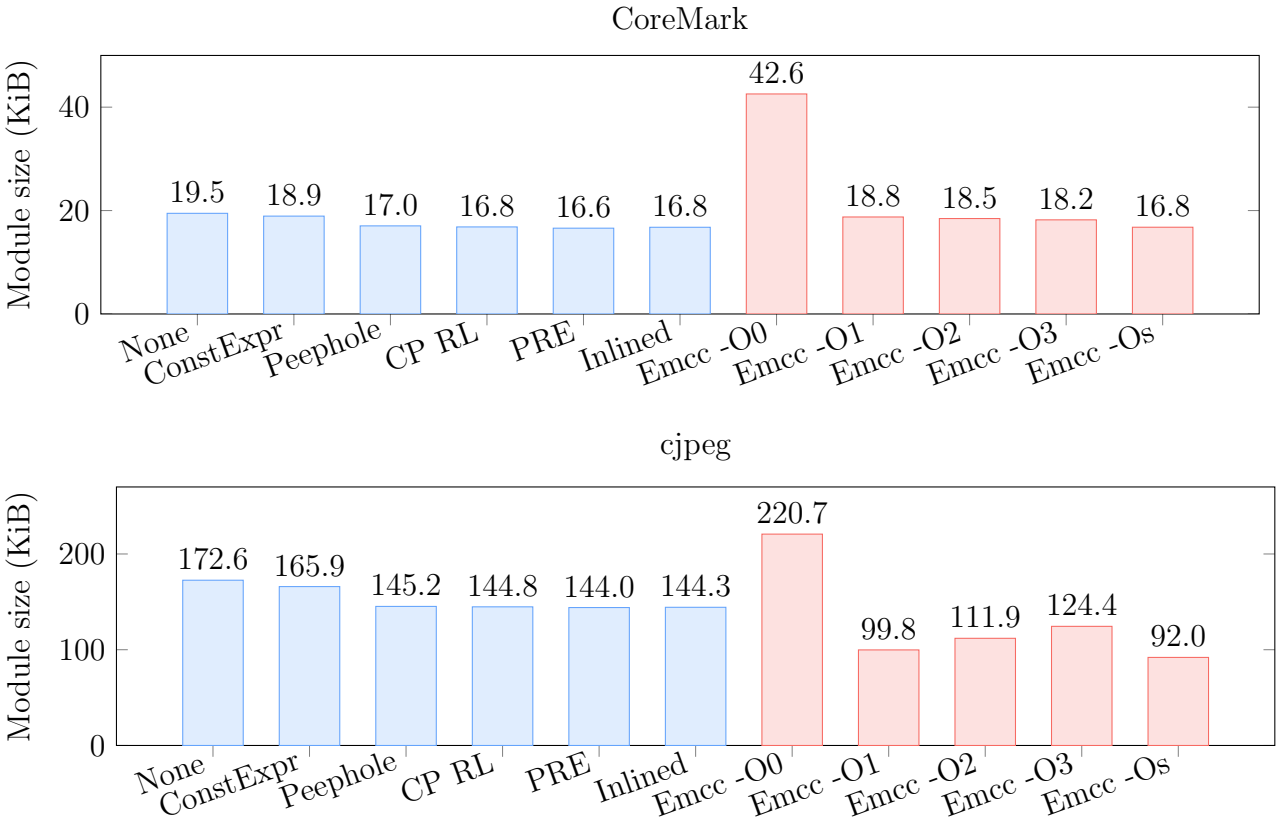
4.1.4 Analysis

My compiler's optimisations operate on the final WebAssembly, limiting it to optimisations that can also be performed by the WebAssembly runtime. Implementing optimisations on intermediate representations would have allowed for more optimisation and allowed the use of information that isn't available in the WebAssembly output. Furthermore, most of my optimisations only consider WebAssembly locals, which limits their usefulness on memory-intensive workloads.

However, my optimisations do offer larger improvements when using simpler WebAssembly runtimes, such as Liftoff. Additionally, my partial-redundancy elimination implementation offers improvements to amenable code even under the sophisticated TurboFan runtime.

4.2 Code Size

Both my compiler and Emscripten produce modules that require runtime JavaScript to enable features like file-system support. The runtime needed by my compiler is significantly smaller but offers fewer features than Emscripten's. For this reason, the below measurements only include the actual WebAssembly modules.



For my compiler, the peephole optimisations offer the largest saving, which makes sense as the targeted instruction patterns are common, and the replacements are always shorter. Some optimisations, such as function inlining, trade increased module size for potentially better performance (unless the original functions can be removed).

My compiler's CoreMark module is practically the same size as Emscripten's module using -Os (optimise for size). libjpeg is substantially larger than CoreMark (around 17K lines of code vs 1500), and accordingly, the generated modules are also larger. Here c2wasm performs less well, and the module's size is between Emscripten's -O0 and -O1.

4.3 Compiler Size

Once bundled using webpack, c2wasm is just 300 KiB of JavaScript, including the (limited) standard library, which is first bundled into a JSON file. Additionally, I created several web demos (see Appendix D) demonstrating c2wasm's web-friendly nature and size. However, c2wasm does require a compatible JavaScript runtime such as Node.js (around 80 MiB) or a modern web browser.

Emscripten is installed using the Emscripten SDK and is around 800 MiB (and includes Node.js). This includes many extra files such as documentation and tests, but the included LLVM binary alone is over 70 MiB, making the compiler unsuitable for the web. However, due to the massive discrepancy in supported features, the sizes cannot be directly compared.

4.4 C Support

In my project proposal, I planned to support a common subset of C89 with some C99 features. This was achieved successfully, with my compiler compatible with the majority of C89 and all the listed C99 features. I also implemented features I planned to exclude, including function pointers (see section 3.8.6), partial `goto` statement support (section 3.8.1) and decent support for the standard library (section 3.9).

This is also demonstrated by the usage of existing libraries and programs. For example, the printf implementation was only modified to add code integrating it into the standard library. In CoreMark, only the configuration files, `portme.c` and `portme.h`, were modified.

However, there are still missing features. Compiling libjpeg required minor changes, primarily to simplify preprocessor macros and some uses of `goto` (see Appendix G.3).

4.5 Correctness

The correctness of my compiler was verified through a suite of 143 automated tests. These were run automatically for each commit using GitHub Actions, on the long-term support version of Node.js (14.x) and the latest version (15.x), on both Windows and Ubuntu. Whilst running the tests, code coverage was also calculated, showing that my tests cover 94% of functions and 93% of the lines of code.

Additionally, both CoreMark and libjpeg pass their automated tests. CoreMark calculates a CRC of the results, which is compared against a predefined value for validation. libjpeg has two JPEG decompression tests, which produce the correct output and one compression test, which produces a valid JPEG output (compression isn't completely deterministic).

4.6 Success Criterion

"The project will be a success if it compiles a single-file program written in a common subset of the C language into WebAssembly and executes it correctly."

The previous two sections demonstrate that my compiler can compile a common subset of the C language (section 4.4) and that the output is correct (section 4.5). Furthermore, my compiler can compile both single-file programs and those utilising multiple files. Therefore, the project's success criterion has been met and exceeded.

Additionally, I have shown that c2wasm can produce WebAssembly with similar performance to an existing open-source tool.

5 Conclusions

I have successfully implemented a C to WebAssembly compiler, which I called c2wasm. The project's success criterion of compiling single-file programs written in a common subset of C was met successfully (see section 4.6) and c2wasm supports compiling large programs across multiple files. The other extensions, adding additional C features and more complex optimisations, have also been implemented. This includes supporting a substantial number of features I had planned to exclude and implementing partial-redundancy elimination, a complex optimisation requiring four data-flow analyses.

I also built multiple demos demonstrating the use of c2wasm inside the web browser (see Appendix D), which is the project's primary use case. My compiler was evaluated against Emscripten, an existing C to WebAssembly compiler, which must be used locally, and c2wasm performs between its -O0 and -O1 optimisation levels.

5.1 Future Work

To extend my compiler, support for C89 could be completed, and more modern standards could be implemented, increasing compatibility. In particular, full `goto` support would be interesting to add, as it would necessitate transforming arbitrary control flow into the structured control flow required by WebAssembly.

Another possibility would be adding a command-line interface and integration with standard tooling such as Makefiles, making it easier to compile existing programs utilising complex build systems. Together with increased C support, this would allow for existing standard-library implementations to be used, removing standard-library support as a limitation for application compatibility. Better IO support could also be implemented using the upcoming WebAssembly System Interface (WASI)[31], which exposes POSIX-like syscalls in WebAssembly modules.

Adding additional optimisations, such as memory optimisations involving pointer analysis, would also be an exciting extension to the project. These could be implemented after the code-generation stage, like the current optimisations, or additional immediate representation (IR) stages could be added, allowing easier and more precise optimisation at the expense of reworking existing parts of the compiler.

5.2 Lessons Learnt

If I were going to repeat this project and had more time, I would build more features into the original plan for the project.

To ensure the success criterion was met and due to the limited time available, more complex optimisations were left as an extension. This made their implementation substantially more difficult, and has demonstrated the value of optimising intermediate representations. Therefore, if I redid the project, I would design it with additional optimisation-friendly IR stages in mind.

Additionally, I would implement integration with existing tooling from the start, which may save time when trying to support the standard library and compile existing libraries and programs.

Finally, this project has demonstrated the benefit of automated testing, which makes it much easier to identify when issues are introduced and to prevent their introduction in the first place. Tests are especially useful to ensure complicated cases work as expected, such as ensuring that `const * ` and `* const` pointers work correctly. Regression tests are also critical and ensure that hard-to-find bugs do not unexpectedly reappear.

Bibliography

[1] *American National Standard Programming Language C, ANSI X3.159-1989*. American National Standards Institute. 1989.

[2] *WebAssembly*. URL: https://webassembly.org/ (visited on 20/02/2021).

[3] *Announcing the Bytecode Alliance: Building a secure by default, composable future for WebAssembly*. URL: https://bytecodealliance.org/articles/announcing-the-bytecode-alliance (visited on 01/05/2021).

[4] *Emscripten*. URL: https://github.com/emscripten-core/emscripten (visited on 20/02/2021).

[5] *Cheerp - a C/C++ compiler for Web applications*. URL: https://github.com/leaningtech/cheerp-meta (visited on 20/02/2021).

[6] *asm.js - an extraordinarily optimizable, low-level subset of JavaScript*. URL: http://asmjs.org/ (visited on 20/02/2021).

[7] *WebAssembly Specification*. Version Release 1.1 (Draft, Feb 18, 2021). URL: https://webassembly.github.io/spec/core/ (visited on 21/02/2021).

[8] *WebAssembly JavaScript Interface*. Version Editor's Draft, 18 February 2021. URL: https://webassembly.github.io/spec/js-api/index.html (visited on 21/02/2021).

[9] *ANSI C Yacc grammar*. URL: http://www.quut.com/c/ANSI-C-grammar-y-1999.html (visited on 27/10/2020).

[10] *zaach/jison: Bison in JavaScript*. URL: https://github.com/zaach/jison (visited on 27/10/2020).

[11] *GNU Bison*. URL: https://www.gnu.org/software/bison/ (visited on 22/02/2021).

[12] *GerHobbelt/jison: bison/YACC/LEX in JavaScript*. URL: https://github.com/GerHobbelt/jison (visited on 27/10/2020).

[13] *ANTLR v4*. URL: https://github.com/antlr/antlr4 (visited on 27/10/2020).

[14] *antlr4ts - TypeScript/JavaScript target for ANTLR 4*. URL: https://github.com/tunnelvisionlabs/antlr4ts (visited on 27/10/2020).

[15] *C and C++ are not context free*. URL: http://trevorjim.com/c-and-cplusplus-are-not-context-free/ (visited on 22/02/2021).

[16] *no-content/moo: Optimised tokenizer/lexer generator!* URL: https://github.com/no-context/moo (visited on 28/10/2020).

[17] *V8 JavaScript engine*. URL: https://v8.dev/ (visited on 21/02/2021).

[18] *Browser Market Share Worldwide*. Version Jan 2020 - Jan 2021. URL: https://gs.statcounter.com/browser-market-share (visited on 21/02/2021).

[19] *WebAssembly compilation pipeline*. URL: https://v8.dev/docs/wasm-compilation-pipeline (visited on 21/02/2021).

[20] *avajs/ava: Node.js test runner that lets you develop with confidence*. URL: https://github.com/avajs/ava (visited on 24/10/2020).

[21] *istanbuljs/nyc*. URL: https://github.com/istanbuljs/nyc (visited on 21/04/2021).

[22] *ESLint - Pluggable JavaScript linter*. URL: https://eslint.org/ (visited on 21/02/2021).

[23] *typescript-eslint/typescript-eslint: Monorepo for all the tooling which enables ESLint to support TypeScript*. URL: https://github.com/typescript-eslint/typescript-eslint (visited on 24/10/2020).

[24] *webpack*. URL: https://webpack.js.org/ (visited on 27/10/2020).

[25] V. K. Paleri, Y. N. Srikant and P. Shankar. "A simple algorithm for partial redundancy elimination". In: *ACM SIGPLAN Notices* 33.12 (1998), pp. 35–43. DOI: 10.1145/307824.307851.

[26] *ECMAScript® 2020 Language Specification*. URL: https://262.ecma-international.org/11.0/#sec-intro (visited on 24/03/2021).

[27] Alon Zakai. "Emscripten: An LLVM-to-JavaScript Compiler". In: *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*. OOPSLA '11. Portland, Oregon, USA: Association for Computing Machinery, 2011, pp. 301–312. ISBN: 9781450309424. DOI: 10.1145/2048147.2048224. URL: https://doi.org/10.1145/2048147.2048224.

[28] *mpaland/printf: Tiny, fast, non-dependent and fully loaded printf implementation for embedded systems*. Version d3b984684bb8a8bdc48cc7a1abecb93ce59bbe3e. URL: https://github.com/mpaland/printf (visited on 23/03/2021).

[29] *musl libc*. Version 1.2.1. URL: https://musl.libc.org/ (visited on 03/04/2021).

[30] *AVR Libc Home Page*. Version 2.0.0. URL: https://www.nongnu.org/avr-libc/ (visited on 03/04/2021).

[31] *WASI: The WebAssembly System Interface*. URL: https://wasi.dev/ (visited on 10/04/2021).

[32] *Implementing Malloc: First-fit Free List*. URL: https://embeddedartistry.com/blog/2017/02/15/implementing-malloc-first-fit-free-list/ (visited on 02/04/2021).

[33] *CPU Benchmark – CoreMark – EEMBC Embedded Microprocessor Benchmark Consortium*. URL: https://www.eembc.org/coremark/index.php (visited on 25/04/2021).

[34] *libjpeg*. Version 6b. URL: http://libjpeg.sourceforge.net/ (visited on 25/04/2021).

Appendices

Appendix A Lines of Code Breakdown

The proforma's line count is computed by combining multiple `cloc` reports using the following commands:

- Counting the number of lines of TypeScript in the source folder:
  `cloc c2wasm/src/ --include-lang TypeScript --report-file Source`
  (The generated parser is written in JavaScript and therefore not included)
- Counting the number of lines of TypeScript in the demos folder:
  `cloc c2wasm/demos/ --include-lang TypeScript --report-file Demos`
- Counting the number of lines of TypeScript in the tests folder:
  `cloc c2wasm/tests/ --include-lang TypeScript --report-file Tests`
  (Some of these lines are C code embedded in tests, not TypeScript)
- Counting the number of lines of C in the standard library implementation:
  ```
  cloc c2wasm/src/c_library/impl/ --exclude-dir libraries \
      --include-lang "C,C/C++ Header" --report-file C_Standard_Library
  ```
  (The excluded `libraries` folder contains source files taken from existing libraries)
- Summing the reports together as follows:
  `cloc --sum-reports "Source" "Demos" "Tests" "C_Standard_Library"`

The output of final command, the summed report, is shown below:

```
github.com/AlDanial/cloc v 1.81
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
TypeScript                     126           2404            712          12952
C                                7            139             25            536
C/C++ Header                    16             74             44            414
-------------------------------------------------------------------------------
SUM:                           149           2617            781          13902
-------------------------------------------------------------------------------


-------------------------------------------------------------------------------
File                         files          blank        comment           code
-------------------------------------------------------------------------------
Source                          63           1476            602           7741
Tests                           55            726             89           4130
Demos                            8            202             21           1081
C_Standard_Library              23            213             69            950
-------------------------------------------------------------------------------
SUM:                           149           2617            781          13902
-------------------------------------------------------------------------------
```

Appendix B C Standard-library Support

The following standard-library headers are supported:
(See `<stdio.h>`*,* `<stdlib.h>` *and* `<string.h>` *for details on the libraries used and their licences.)*

- `<assert.h>` *Assert macro* *Unsupported*
- `<ctype.h>` *Character-testing functions* *Supported*

  e.g. `isdigit()`, `toupper()`
- `<errno.h>` *Error codes* *Partially supported*

  Includes the necessary definitions for C89, but `errno` is never set by the standard library.
- `<float.h>` *Implementation-defined floating-point limits* *Supported*
- `<limits.h>` *Implementation-defined integer limits* *Supported*
- `<locale.h>` *Localisation functions* *Unsupported*
- `<math.h>` *Mathematical functions* *Partially supported*

  Only functions that have corresponding WebAssembly instructions are implemented using inline WebAssembly (see section 3.8.7): `sqrt`, `ceil`, `fabs` and `floor` from C89 and `sqrt`, `ceil`, `fabs` and `floor` from C99.
- `<setjmp.h>` *Non-local jumps* *Unsupported*

  Used to provide jumping between functions, which would be an extension after providing full `goto` support.
- `<signal.h>` *Signal-handling functions* *Unsupported*

  WebAssembly normally runs within an embedding host environment that handles any received signals.
- `<stdarg.h>` *Variable argument lists* *Supported*

  Macros for accessing the extra arguments in variadic functions (see section 3.8.7)
- `<stdbool.h>` *[C99] Boolean data type* *Supported*

  Defines `true`, `false` and aliases `_Bool` to `bool`
- `<stddef.h>` *Basic types and macros* *Supported*

  Defines `NULL`, `size_t`, `ptrdiff_t` and `offsetof`.
- `<stdint.h>` *[C99] Exact-width integer types* *Supported*

  e.g. `uint32_t`, `int64_t`

- `<stdio.h>` *IO functions* *Partially supported*

  See section 3.9.1 for details.

  The implementation of the `printf` family of functions was taken from the mpaland/printf GitHub repository[28] (licensed under the MIT Licence).

- `<stdlib.h>` *Utility functions* *Supported*

  The `strtod`, `strtol`, `strtoul`, `bsearch` and `qsort` functions are taken from AVR Libc[30] (licensed with a Modified BSD Licence) and slightly modified.

  The `malloc` implementation is based on an implementation by Embedded Artistry[32], see section 3.9.2 for details.

- `<string.h>` *String functions* *Supported*

  All the string functions are taken from musl[29], licensed under the MIT Licence, with only minor modifications. Some non-standard BSD string functions defined in `<strings.h>` are also supported as they were included in musl.

- `<time.h>` *Time and date functions* *Partially supported*

  Only the `clock()` function and `clock_t` typedef are supported.

The licences used by the open-source libraries are compatible with each other and with c2wasm's MIT Licence. To comply with the licences, the licences must be retained with the source code, and binaries must reproduce the licences in materials provided with the distribution. See the individual licences, included with the project's source code, for their full copyright notices, lists of conditions and disclaimers.

Appendix C Optimisation Flags

The optimisation flags control which optimisations are performed and can be changed at runtime using the `setFlags({...} | "none" | "default")` function. The following table details every c2wasm optimisation flag, the default value (second column) and the value in each benchmark configuration used in the evaluation (final group of columns).

Flag name	Defaults	None	ConstExpr	Peephole	CP RL	PRE	Inlined
`generation_try_constant_expr` In the code generation stage, try evaluating each expression as a constant. If successful, generate a constant value instruction instead of the expression.	✓	✗	✓	✓	✓	✓	✓
`generation_zero_shadow_stack` Insert code to clear the shadow stack before using it. [For debugging, not normally needed]	✗	✗	✗	✗	✗	✗	✗
`generation_switch_br_table` Replace the default chain of comparisons with a single `br_table` instruction if the range of case values is small. [Not fully tested, but doesn't seem to impact performance]	✗	✗	✗	✗	✗	✗	✗
`peephole_local_tee` `local.set [idx]; local.get [idx]` → `local.tee [idx]`	✓	✗	✗	✓	✓	✓	✓
`peephole_i32_constants_ops` `i32.const [a]; i32.const [b]; i32.[op]` →`i32.const [a op b]`	✓	✗	✗	✓	✓	✓	✓

`peephole_constants_add_mul`
`T.const [a]; T.const [b]; T.add` → `T.const [a + b]`
`T.const [a]; T.const [b]; T.mul` → `T.const [a * b]`
	✓	✗	✗	✓	✓	✓	✓

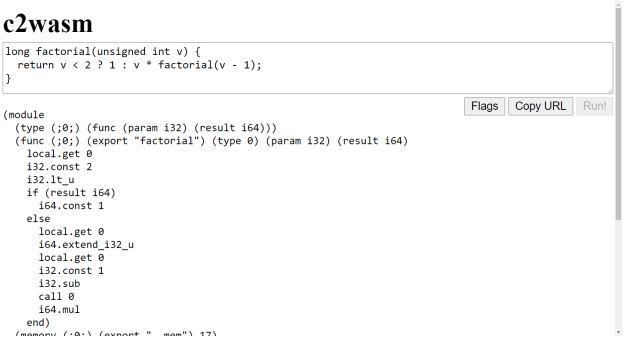
`peephole_add_0` Remove useless `T.const 0; T.add` instructions	✓	✗	✗	✓	✓	✓	✓
`peephole_combine_adds` `i32.const [a]; i32.add; i32.const [b]; i32.const` → `i32.const [a + b]; i32.add`	✓	✗	✗	✓	✓	✓	✓
`peephole_load_offset` `i32.const [a]; i32.add; T.load(offset=[b])` → `T.load(offset=[a + b])`	✓	✗	✗	✓	✓	✓	✓
`peephole_unused_blocks` Replace blocks and loops that are never branched to.	✓	✗	✗	✓	✓	✓	✓
`peephole_constant_if` Replace if instructions with constant conditions.	✓	✗	✗	✓	✓	✓	✓

Flag name	Defaults	None	ConstExpr	Peephole	CP RL	PRE	Inlined
`copy_propagation` Enables the copy propagation and constant propagation pass. (See section 3.10.5)	✓	✗	✗	✗	✓	✓	✓
`reallocate_locals` Enable the reallocation of WebAssembly locals as an optimisation pass and disable the code-generation stage reusing locals. (See section 3.10.6)	✓	✗	✗	✗	✓	✓	✓
`live_range_splitting` Enables the splitting of WebAssembly locals before the reallocation pass. (See section 3.10.6)	✓	✗	✗	✗	✓	✓	✓
`unused_locals` Simple optimisation pass that removes any locals which are no longer used.	✓	✗	✗	✗	✓	✓	✓
`dead_code_elimination` Enables the dead-code elimination pass. (See section 3.10.2)	✓	✗	✗	✗	✓	✓	✓
`peephole_2nd_pass` Enable an additional pass of peephole optimisations after the other intraprocedural optimisations.	✓	✗	✗	✗	✓	✓	✓
`partial_redundancy_elimination` Enables the partial redundancy elimination (PRE) pass. (See section 3.10.7)	✓	✗	✗	✗	✗	✓	✓
`inlining` Enables the interprocedural function-inlining pass. (See section 3.10.8)	✗	✗	✗	✗	✗	✗	✓

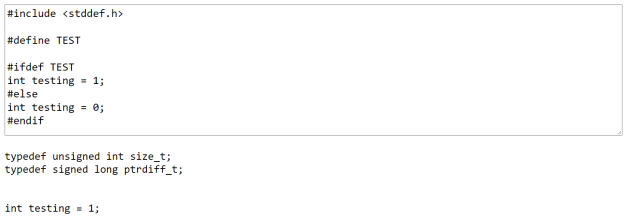
Appendix D Web Demos

The demos are built using the `npm run build` command, which uses webpack to bundle the source files in the `demos/` folder into HTML files and JS bundles for each demo in the `dist/demos` folder. GitHub Actions was used to automatically build and deploy the demos to GitHub Pages for each commit. Below is a list of the demos included with c2wasm:

`index.html`: Full demo of the compiler

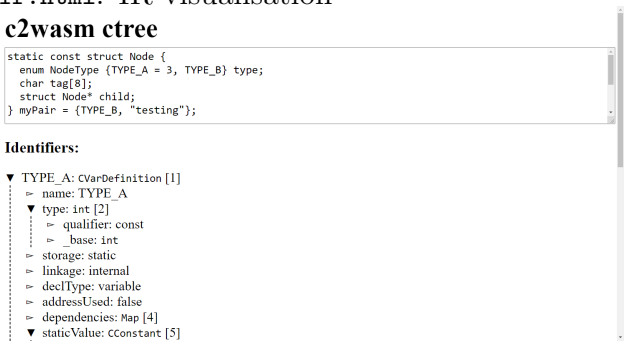


`preprocessor.html`: Preprocessor demo

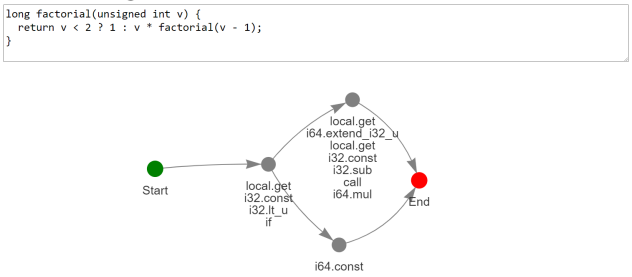


`parsetree.html`: Parse tree visualisation

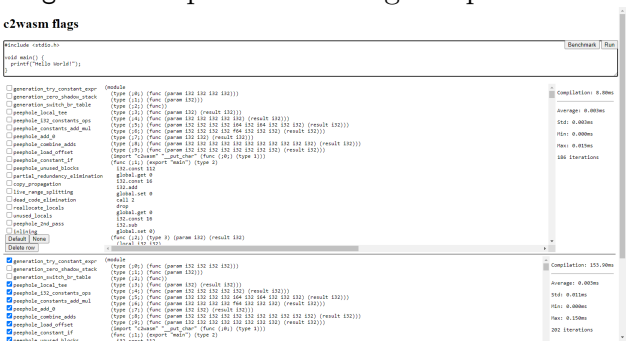
`ir.html`: IR visualisation



`cfg.html`: Control-flow graph visualisation



`flags.html`: Optimisation flag comparision



`CoreMark.html`: In-browser compilation and demo of the CoreMark benchmark

`JPEG.html`: In-browser compilation and demo of libjpeg

Appendix E File System Emulation Example

The following TypeScript shows how the runtime wrapper and the c2wasm API can be used to run a program that reads in files and then produces an output file.

```
const source = new Map<string, string>();
source.set("main.c", "...");
source.set("lib.c", "...");
source.set("lib.h", "...");

const initialFs = new Map<string, Uint8Array>();
initialFs.set("hello.txt", new TextEncoder().encode("Hello world!"));

const files = new c2wasm.runtime.Files(
    /* output function */ (char) => process.stdout.write(char),
    /* input function  */ () => "" /* no standard input available */,
    /* initial files   */ initialFs);

// compile the module
const module = await c2wasm.compile(source, { // define preprocessor macros
    FILES: "1"  // enable FS emulation support
}).execute({ // instantiate the module
    c2wasm: {...files.getImports()} // using the FS wrapper imports
});

// run the module
c2wasm.runtime.mainWrapper(module, /* main(...) arguments */ ["main"]);

// access the created output file
const result = files.getContents("output.txt");
```

Appendix F PRE Example

Below is the source code for the toy partial-redundancy elimination (PRE) benchmark. The function `f` was created to demonstrate the PRE implementation and is otherwise meaningless.

```
1  static long f(long x) {
2    long y;
3    if (x % 10 == 0) {
4      y = x * x * x;
5    } else {
6      y = 1L << (x % 10);
7    }
8    y ^= x;
9    return y / (x * x * x);
10 }
11
12 int main() {
13   int result = 0;
14   for (int i = 10; i <= 1000 * 1000; i++) result += f(i);
15   return result;
16 }
```

The full WebAssembly produced for the `f` function with and without PRE is shown on the following page.

Without PRE:

```
(func (param i64) (result i64)
    (local i64)
    local.get 0 // x % 10 == 0
    i64.const 10
    i64.rem_s
    i64.const 0
    i64.eq
    if
      local.get 0 // y = x * x * x
      local.get 0
      i64.mul
      local.get 0
      i64.mul
      local.set 1
    else
      i64.const 1 // 1 << (x % 10)
      local.get 0
      i64.const 10
      i64.rem_s
      i64.shl
      local.set 1
    end
    local.get 1 // y ^= x
    local.get 0
    i64.xor
    local.get 0 // y / (x * x * x)
    local.get 0
    i64.mul
    local.get 0
    i64.mul
    i64.div_s)
```

With PRE:

```
(func (param i64) (result i64)
    (local i64 i64)
    local.get 0
    i64.const 10
    i64.rem_s
    local.tee 1 // save x % 10
    i64.const 0
    i64.eq
    if
      local.get 0
      local.get 0
      i64.mul
      local.get 0
      i64.mul
      local.tee 1
      local.set 2 // save x*x*x
    else
      i64.const 1
      local.get 1 // load x % 10
      i64.shl
      local.set 2
      local.get 0 // PRE inserted
      local.get 0 // computation
      i64.mul
      local.get 0
      i64.mul
      local.set 1 // save x*x*x
    end
    local.get 2
    local.get 0
    i64.xor
    local.get 1 // load x * x * x
    i64.div_s)
```

The PRE optimisation pass does two things:

1. The fully redundant computation of `x % 10` in the else branch has been removed. The value computed in the `if` condition is saved and used again to avoid recomputation.
2. The computation of `x * x * x` for the return value was partially redundant due to its computation in the `if` branch. Therefore, an extra computation was inserted into the `else` branch and both branches save the value calculated. When calculating the function's return value, this saved value is then loaded, preventing the possible recalculation.

The evaluation chapter (see section 4.1.1) shows that these changes improve performance by around 30%.

Appendix G Benchmark Details

All the benchmarks were run automatically using a custom test framework in the `tests/benchmark/` folder. Each benchmark was run in each configuration 100 times, and then the averages and standard deviations were calculated. Additionally, the framework handled measuring the size of the generated WebAssembly module for the code size evaluation.

Appendix G.1 PRE Benchmark

The toy PRE benchmark was custom written as described in Appendix F.

Benchmark command	`npm exec -- ts-node tests/benchmark/run.ts toy`
Benchmark runner	`tests/benchmark/toy.ts`
Benchmark source	Embedded in the runner.

Appendix G.2 CoreMark

The CoreMark source files are licensed under the Apache Licence, Version 2.0. Only the configuration files, `portme.c` and `portme.h`, were modified.

Benchmark command	`npm exec -- ts-node tests/benchmark/run.ts coremark`
Benchmark runner	`tests/benchmark/coremark.ts`
Benchmark source	`tests/benchmark/coremark/`
CoreMark parameters	2K Performance run parameters (0x00, 0x00, 0x66)
GCC Flags	`-DCOMPILER_FLAGS="" -DPERFORMANCE_RUN=1 -DITERATIONS=0`
Emcc Flags	`-DCOMPILER_FLAGS="" -DPERFORMANCE_RUN=1 -DITERATIONS=0`

Appendix G.3 libjpeg

Version 6b of libjpeg was used and is licensed under a custom permissive (BSD-like) licence.

Benchmark command	`npm exec -- ts-node tests/benchmark/run.ts jpeg`
Benchmark runner	`tests/benchmark/jpeg.ts`
Benchmark source	`tests/benchmark/jpeg/`
GCC Flags	`-w`
Emcc Flags	`-w -s EXIT_RUNTIME=1 -s NODERAWFS=1`

Modifications:

Due to c2wasm's limited preprocessor, which was an extension to the project, macros split across multiple lines needed line continuing `\` symbols inserted at the end of each line. All other modifications were marked with `// CHANGED [reason]` comments in the source code:

- `cjpeg.c` was modified to add timing support for the benchmark
- `jconfig.h` (the configuration file) was modified.

- `jdhuff.c`, `jdhuff.h` and `rdswitch.c` were modified to make the `goto` statements compatible with c2wasm.
- Extra (unreachable) return statements were added to two functions in `jdmarker.c`, as c2wasm didn't think the functions always returned.

This shows that only minor changes were required to get a relatively large (17K lines of code) code base running through c2wasm.

Appendix G.4 System Configuration

Operating System	Debian 10 running on Windows 10 (20H2) using WSL 2
CPU	AMD Ryzen 9 5900X
RAM	32GB DDR4 3200MHz CL16
Node.js Version	v16.0.0
Emscripten Version	emcc version 2.0.18, clang version 13.0.0
GCC Version	v8.3.0

C Compiler to WebAssembly

Project Supervisor Dr Timothy Jones

Director of Studies Dr John Fawcett

1 Introduction and Description of the Work

The aim of my proposed project will be to create a compiler from a subset of C targetting WebAssembly. I plan to create this compiler using TypeScript, enabling both the compiler and its WebAssembly output to run inside modern web browsers as well as Node.js.

WebAssembly (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine, designed as a portable compilation target[1]. It is supported in all modern browsers and is sandboxed, only being able to access the predefined imports and supplied function pointers. This makes it ideal for both integrating existing (non JavaScript) code into web applications and for speeding up complex calculations achieving near-native speed.

I plan to use an existing library/tool (such as antlr4ts[2]) to help create a lexer and parser for a modified C grammar, which my compiler will then process through semantic analysis and one or more intermediate forms. A few basic optimisations will be applied and it will then output WebAssembly. It will also supply some simplified versions of the most common C standard library functions, which I plan to implement in WebAssembly or another simple language which can easily target WebAssembly.

2 Starting Point

C – I have used C in some small projects such as part of my Part IB Group Project after learning it for the Part IB Programming in C and C++ course.

TypeScript – I learnt TypeScript for an internship this summer, so have a decent understanding and prior to this I only had very basic JavaScript knowledge. This also gave me some more insight into the some of the tools that I plan to use.

Compilers – I have no prior experience writing a complete compiler but I did take the Part IB Compilers course and plan to take the Part II Optimising Compilers course.

WebAssembly – Before September I had very little knowledge of WebAssembly and in the run up to this project I have read (parts of) the WebAssembly specification, written some basic programs (e.g. factorial) in WebAssembly and investigated creating and using WebAssembly from JavaScript/TypeScript.

I have also started looking into which lexers and parsers would be suitable for the C grammar, however neither lexing nor parsing is the focus of this project.

3 Substance and Structure of the Project

The substance of the project is compiling the parse tree provided by the parser down into low-level WebAssembly. This involves processing it into a useable abstract syntax tree (AST), then into one or more intermediate representations before generating the WebAssembly code. I plan to implement a few basic optimisations (such as replacing `local.set` followed by `local.get` by `local.tee` which is like `local.set` but leaves its argument on the stack) which may transform the intermediate or final WebAssembly forms.

My plan is to compile a C-like language, mostly stemming from C89 (the original standardised version of C) but with some more modern features, for example allowing variable declarations anywhere inside a block from C99. Some features of C would add substantial complexity outside the scope of this project and will therefore be excluded. However, I hope to support enough C to make this project viable for running real code. Problematic C features that I have already identified and plan to exclude from my target language include:

- My initial subset of C will only compile a single source file and will not support linking. This means that the `extern` keyword will not be supported. Futhermore, I will not be including support for the C preprocessor or any of its directives.
- Function pointers will not be supported as to be implemented in WebAssembly they have to be an index into the (optional) function table meaning they are not strictly memory addresses (unlike normal pointers) and require extra handling. I hope to implement normal pointers as close as possible to the C standard.
- The C Standard Library will not be supported with the exception of some common functions such as `malloc` and `free`, for which I will provide simplified implementations (e.g. potentially never reclaiming allocated space). Futhermore I hope to provide some basic maths functions such as square root for which there are native WebAssembly instructions.
- The C grammar is not context free, making it harder to parse, due to having to differentiate between identifiers and `typedef` names based on previous definitions. Further complicating this is that these declarations are scoped and can be overridden locally so initially I plan to exclude `typedef` support. However, it should be possible to only support simple top level `typedef` declarations (and not allow overriding once defined) which would make the most common use case work.
- The `goto` keyword will not be supported as it is rarely used and commonly viewed as bad practice. Furthermore, in C `goto` can be used to jump anywhere within the current function, whereas WebAssembly only provides the ability to jump to the start or end of certain blocks so allowing arbitrary jumping would add substantial complexity.
- The `register` and `volatile` keywords will not be supported as WebAssembly does not allow access to such low-level hardware features.
- Unlike normal C, the `main` function will not be automatically run to start the program due to how WebAssembly is designed to run alongside JavaScript in the browser.

Some of the more complicated features include supporting `struct` and `union` as well as allowing these potentially large structures on the stack. Allowing this will require investigation and may involve having to use part of the memory space as an additional stack for larger values which would add extra complexity.

To ensure the project is working as planned, I will use a combination of smaller unit tests and automated end-to-end tests that use the project to compile some provided C and check that it outputs the correct value.

To evaluate my project I intend to compare my compiler against one or more common C to WebAssembly compilers for a few demo C programs (within the above defined language). This will include performance of the compiled WebAssembly when running inside Node.js, potentially as well as other measures such as binary size (though this may be difficult due to the inclusion of different debugging information). I also plan to compare the performance against the same algorithms written in JavaScript to see if my compiler could be used to optimise hot methods within JavaScript code.

4 Success Criterion

The project will be a success if it compiles a single-file program written in a common subset of the C language into WebAssembly and executes it correctly.

When evaluating the project I hope my compiler can produce code which outperforms the same algorithms implemented in JavaScript, however this will be dependent on how well existing runtimes perform optimisations whilst interpreting JavaScript. I expect that existing C compilers may outperform my compiler for many reasons including the differences in development timescales and number of developers but they will be a useful comparison to see what can be achieved and how close I can get.

4.1 Potential Extensions

A potential extension for this project is allowing multiple files to be compiled (at the same time) and linked together which would greatly expand the amount of real code which could be compiled by this project. This would include supporting some C preprocessor directives, either by implementing support for an existing C preprocessor or by writing my own which only supports the most basic directives.

Other possible extensions include tackling other excluded features (such as adding limited function pointer support) or adding more advanced optimisation techniques.

5 Plan of Work

5.1 17th Oct – 6th Nov 2020 (3 weeks)

- Researching C (in particular the C89 and C99 standards) and defining more exactly the features I intend to support.
- Deciding on a lexer and parser and setting them up to produce a parse tree.
- Perform some transformations on the parse tree to make it more useful (an abstract syntax tree).

Milestone: Demo program which can take any C within my initial subset and produce an AST.

5.2 7th Nov – 20th Nov 2020

- Start work on processing the AST and transforming it into an intermediate form.
- Reading the WebAssembly binary specification and creating helpers to help with producing the WebAssembly output. *I believe it is best to do this early to give me a better idea of what I am targetting as I work through the rest of the project.*

Milestone: Wasm helper functions should output a working WebAssembly Module when tested against small functions (e.g. factorial) written in Wasm.

5.3 21st Nov – 4th Dec 2020

Note: NLP deadline on 4th worth 80%, therefore this is a smaller unit of work.

- Continue with processing/transforming the AST.
- Write some tests to ensure existing AST processing and WebAssembly helper functions work.

Milestone: Automated testing for parsing and Wasm helpers.

5.4 5th Dec – 18th Dec 2020

- Basic code generation (translating the intermediate form into WebAssembly) for functions and basic expressions.

Milestone: Correct compilation of very simple C functions which simply return a simple expression into WebAssembly which executes correctly.

5.5 19th Dec – 1st Jan 2021

- Code generation support for simple variables and control flow such as `if` statements, loops and function calls.
- Research and begin work into supporting more complex types (e.g. pointers).

Milestone: Correct compilation of simple C programs, supporting functions, variables, control flow and loops.

5.6 2nd Jan – 15th Jan 2021

- Finish implementation of pointers.
- Start work on handling `struct`s, including writing simple `malloc` and `free` functions.

Milestone: Correctly compilation of simple C code utilising pointers.

5.7 16th Jan – 29th Jan 2021

- Finish handling `struct`s including implementing `union`.
- Begin planning the progress report.

Milestone: **Success Criterion completion**
My project should now meet the the defined success criterion and be able to compile single-file programs within my C-like language.

5.8 30th Jan – 12th Feb 2021

- Finalise the progress report.
- Implement some more common and mathematical C library functions.
- Prepare for the progress report presentation.

Deadline: Progress Report – 5th Feb
Progress Report Presentations take place between 11th - 16th Feb

5.9 13th Feb – 26th Feb 2021

- Implement some basic compiler optimisations.
- Testing and bug fixing.
- Start working on an extension if time permits.

Milestone: Multiple compiler optimisations should be implemented and documented.

5.10 27th Feb – 12th Mar 2021

- Final bug fixes and program optimisations for the core implementation.
- Further extension work if time permits.
- Start work on the dissertation document, including writing the introduction section.

Milestone: Dissertation introduction section written.

5.11 13th Mar – 26th Mar 2021

- Write the preparation section.
- Begin the implementation section.

Milestone: Dissertation preparation section written.

5.12 27th Mar – 9th Apr 2021

- Finish the implementation section.
- Write the evaluation section.

Milestone: Dissertation implementation and evaluation sections written.

5.13 10th Apr – 23rd Apr 2021

- Write the conclusions section.

Milestone: **First draft of Dissertation**
The project and dissertation should now be complete.

5.14 24th Apr – 14th May 2021 (3 weeks)

- Proofreading the dissertation.
- Submission in the week leading up to the final deadline.

Deadline: Dissertation and Source Code – 14th May

6 Resource Declaration

I plan to use my own desktop (Intel i7-8700K, 32GB RAM and 1TB SSD) for this project. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.

In case of failure I can instead use my laptop (Intel i7-8550U, 16GB RAM and 1TB SSD) and if this is also not possible I will use the provided MCS machines.

For revision control, I will use Git for both the LaTeX for the dissertation as well as the project source code, which will also be pushed to GitHub. Futhermore my desktop automatically performs hourly offsite backups to the cloud.

I intend to develop the project in TypeScript using JetBrains WebStorm and if this is not possible I will use another compatible IDE such as Microsoft VS Code. Node.js and Chromium will also be used for running and debugging the project but alternatively Mozilla Firefox can be used. The dissertation will be written in VS Code however if needed I can use any text editor. All of these programs can be run on the MCS machines and are either being preinstalled or can be run without installation.

7 References

[1] WebAssembly. `https://webassembly.org/`. Accessed: 13/10/2020.

[2] antlr4ts - TypeScript/JavaScript target for ANTLR 4. `https://github.com/tunnelvisionlabs/antlr4ts`. Accessed: 15/10/2020.